A Framework for Logical Structure Extraction from Software Requirements Documents

by

Rehan Rauf

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

General purpose rich-text editors, such as MS Word are often used to author software requirements specifications. These requirements specifications contain many different logical structures, such as use cases, business rules and functional requirements. Automated recognition and extraction of these logical structures is necessary to provide useful automated requirements management features, such as automated traceability, template conformance checking, guided editing and interoperability with sophisticated requirements management tools like Requisite Pro. The variability among instances of these logical structures and their attributes poses many challenges for their accurate recognition and extraction. The thesis provides a framework for the extraction of logical structures from software requirements documents. The framework models information about style, structure, and attributes of the logical structures and uses the defined meta-model to extract instances of logical structures. A meta-model also incorporates information about the variability present in the instances. The framework includes an extraction tool, ET, that reads the meta-model and extracts instances of modelled logical structures from the documents. The framework is evaluated on a collection of real-world software requirements documents. Using the framework, different logical structures can be extracted with high precision and recall, each close to 100%. The performance of the extraction tool is acceptable for fast extraction of logical structures from documents with extraction times ranging from a few milliseconds to a few seconds.

Acknowledgements

I would like to thank

Dr. Krzysztof Czarnecki	for his valuable guidance and support throughout the course of my Masters
Dr. Charlie Clarke & Dr. Daniel Berry	for their worthy comments and suggestions that helped improve this thesis
Dr. Michal Antkiewicz	for countless useful discussions and for his guidance throughout the course of this research
My lab mates	for providing a great atmosphere to work in, for their support, and for the great time I have had at Generative Software Development Lab
The cleaning lady	for cleaning the lab every Tuesday and making it a workable environment
My friends in Canada	for their immense help, support, and all the good times
Sarah Ahmad & Zeeshan Malik	for proof reading this thesis

Thank you all !

To my parents.

"It would be possible to describe everything scientifically, but it would make no sense; it would be without meaning, as if you described a Beethoven symphony as a variation of wave pressure."

– Albert Einstein

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Requirements management tools have been developed to allow capturing and managing software requirements specifications (SRS) at the level of logical structures such as use cases, business rules, functional requirements and UI mockups [1]. By knowing the internal attributes of these structures, the tools can offer many advantages including fine-grained traceability, guided (structured) editing and template conformance checking [2]. Despite the availability of such tools, many organizations still utilize general-purpose text editors, such as MS Word, to write SRS. To bridge the gap between structured requirements specifications and free text, tools like Requisite Pro [3] provide capability to import entire documents and link to requirements in the documents. The import process requires parsing the documents to identify requirements contained in them. Current methods used to parse the documents require users to specify either regular expressions or delimiter sequences for every requirement to be extracted. These methods are limited to identifying atomic requirements, in the form of sentences or paragraphs and are insufficient for the extraction of logical structures with multiple attributes.

Several tools have also been developed that offer automated analyses of requirements statements [4, 5, 6]. Whereas these tools use semantically rich techniques to reason about validity of requirements, they do not deal with locating these logical structures. We believe that to run a deeper semantic analysis, it is important to first recognize semantic structures in rich text.

The thesis presents a framework for the extraction of logical structures from rich-text documents written using general-purpose editors. The research was conducted in the following three phases:

1. We collected various requirements documents from different sources including industry.

2. We analyzed a randomly selected subset of the collection of documents from our collection to identify requirements for logical structure extraction and develop our approach.

3. We evaluated our developed approach on the entire collection of documents.

The analysis of the requirements documents shows that logical structures produced for a given project follow a particular template either consistently or with minor variations. A project in this context refers to a system (or a tool) to be implemented or modifications/updates made to an already implemented system. We hypothesize that logical structures can be extracted by finding elements in the documents that match a given template. In the framework, a meta-model for logical structures is created that captures information about attributes (Name, ID, etc.), style (bold, italic etc.) and skeletons (tables, lists etc.) that are used to populate instances of the logical structure. Variations in the attribute identifiers, styles, skeletons and cardinality of attributes are also captured in the meta-model. The meta-model is used by a generic extraction tool, ET, to search for elements in the document satisfying the meta-model. The satisfying elements are extracted as instances of the logical structure represented by the meta-model.

To measure the applicability, effectiveness and efficiency of our approach in extracting logical structures, we created meta-models for several logical structures and used them to extract instances. The results show precision and recall, each close to 100% for all logical structures and extraction time of less than 2 seconds for most logical structures.

The proposed framework provides opportunity for improvement in import and traceability features provided by requirements management tools, allowing users to take advantage of those tools together with a free environment of a generic and already widely adopted text editor . The motivation for such a framework is driven by several key observations.

1.1 Motivation

In many organizations, Word documents are still the preferred way of capturing requirements, which effectively deprives these organizations of many advanced features such as fine grained traceability between related parts of requirements and traceability from the requirements to code, semantic query over the requirements, and support for consistency management in methodologies of writing requirements. Some of the key challenges faced by users of such systems that we identified in a recent consulting engagement [1] are discussed

[1] 40 software architects, requirements analysts, testers and project managers from a large company in Canada were interviewed about common software engineering problems faced by the organization. The interviewee were also asked to rank a set of proposed solutions that may solve the software engineering problems faced by them. [7]

below:

1.1.1 Fragmentation of Documentation

Software specifications often contain certain logical structures prescribed by specification methodologies such as Rational Unified Process[8]. Requirements are typically fragmented over many documents. As a result, many logical structures that are spread across multiple documents reference each other (usually by means of an ID) to completely capture a particular requirement. Reading these requirements is particularly challenging for people new to a project or even for those associated with it for years. Upon encountering a reference, the reader has to locate the document that contains the referenced structure, shift their focus to that document, read and understand the target artifact and then move their focus back to the first document. Such a scenario becomes even more dreadful when the logical structures use references recursively. Reading requirements in such a manner causes the reader to lose context by switching between documents and wastes precious time. This challenge could be addressed by automatic linking of the reference to the target structure, which in turn could be used for creation of sliced views of the documentation. This is not possible unless the logical structures, their attributes, and their boundaries within the documents can be recognized.

1.1.2 Inconsistency in Methodology Across Projects

There are many software projects under development and maintenance in any large organization at any given time. Consistency in the way requirements and other software specifications are written is critical for efficient collaboration among different teams. Organizations that use Word documents to author requirements often create templates for writing logical structures such as use cases and business rules. Such templates, however, cannot be used for automatically checking the conformance of the document with the template later. Significant manual effort is required to ensure the conformance of the document to the specified template. This challenge could be addressed by automatic recognition of logical structures during document editing and providing guidance based on template conformance checking. Similar support is well known in IDEs as *code completion* and *quick fix*.

1.1.3 Manual Import of Legacy Documents

If an organization decides to start using a sophisticated requirements management system like DOORs [9], it may choose to import some of the existing (and at this point legacy)

documents into the new system. Such an import is entirely manual, very time and effort consuming, and consequently rarely undertaken. This challenge could be addressed by automatically translating already recognized logical structures into the format understandable by the tool. Although, verifying that the structures are correctly recognized still requires manual effort, it is significantly smaller than before.

1.1.4 Querying Documents

The need to query requirements documents arrises from time to time. If attributes of logical structures are recognized within the documents, queries can be made using these attributes. For example, a query can produce all the use cases in which actor X appears. Such support can be built into the word processing tools as an enhanced search plug-in that uses the template to identify attributes and allows the user to query those attributes.

To summarize the motivations, we want to preserve the convenience of using well-known editors and enhance the experience by offering artifact completion, linking, displaying definitions for references, and other features well known from IDEs. We believe that the proposed system will be a necessary step towards the realization of this goal.

1.2 Problem Statement

The survey of requirements documents showed that different projects use different templates to write the same logical structures. However, logical structures conform mostly to a single template within the scope of the same project. This provides an opportunity to extract logical structures through searching for a template. Nonetheless, there are many challenges involved in doing so. A template consists of two different parts – the methodology and the physical presentation. **Methodology** refers to attributes of the logical structure and the rules relating to these attributes. For example, use case in one methodology may have id, name, scenario and preconditions as attributes where preconditions may be an optional attribute. **Physical presentation** describes how these structures materialize in the actual documents. For example, a use case may be written using a table or a section containing lists. The following example shows two different methodologies and presentations of use cases.

Fig.1.1a and 1.1b show examples of use cases belonging to two separate systems[2]. The presentations used to write use cases in both are clearly different from one another. In

[2]The examples do not show actual data from collected software requirements documents. However, these examples depict the variability observed in those requirements documents.

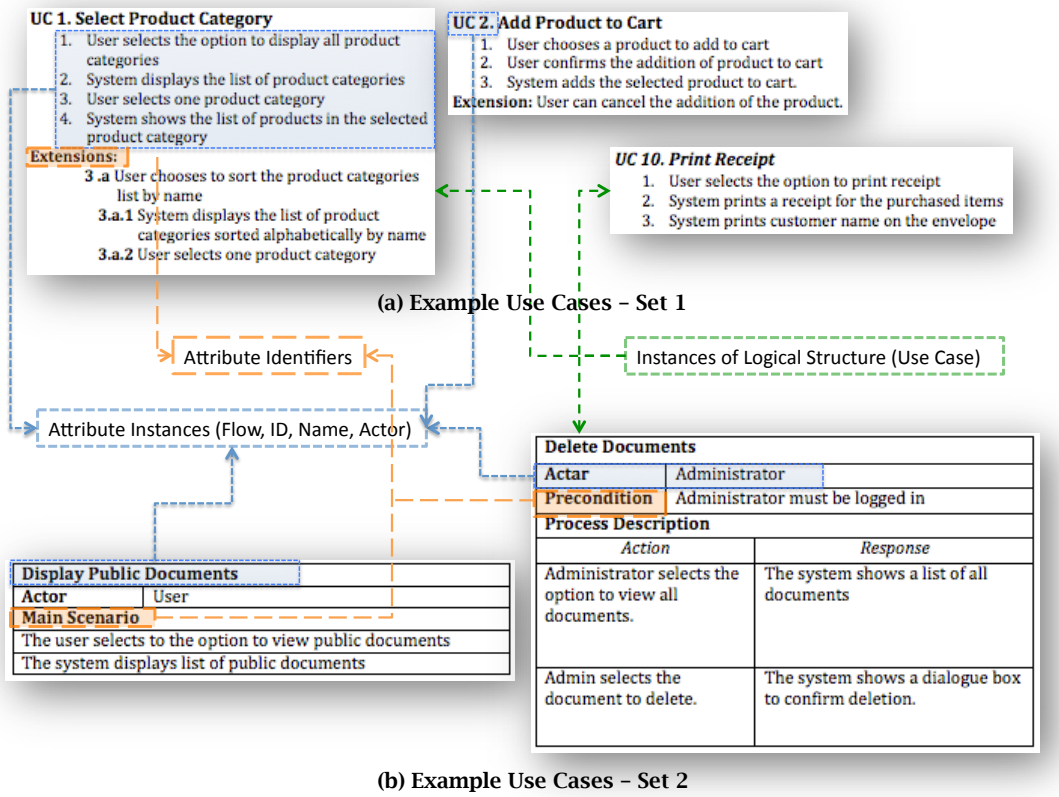


Figure 1.1: Example Use Cases

Fig.1.1a the use case is written as a section of text where as in Fig.1.1b, tables are used to write the use cases. The difference in the attributes is also evident. Use cases in Fig.1.1a use ID, Name, a list for Flow and a sub-section for Extensions where as the use cases in Fig.1.1b use Name, Actor, Precondition, Process Description etc. These differences in attributes stem from using different templates, such as [10, 11] for writing use cases.

Examples of variations within the instances of logical structures are depicted in both Fig.1.1a and Fig.1.1b. These variations can be divided into two categories – **logical variations** and **physical variations**. The logical variations include differences in the number of attribute instances within an instance of a given logical structure, for example, the attribute Extensions is missing in the third instance of use case shown in Fig.1.1a. The logical variations also include using different attributes to represent the same concept, for example use cases in Fig. 1.1b use either a step by step description of the main scenario or description divided into actions and responses.

The physical variations can be further divided into two categories – **accidental variations** and **designed variations**. Accidental variations refer to minor mistakes that happen while writing instances of a given logical structure. These include spelling mistakes, for example "Actar" instead of "Actor" in the second instance shown in Fig.1.1b and mistakes in applying the style(bold, italic and font size), for example unnecessary italicized section title in the third instance shown in Fig.1.1a. The designed variations are physical variations that are allowed by the template, for example use cases in Fig.1.1a either specify the "Extensions" as a sub-section or "Extension" as block of text. Another example of designed physical variation is the convention used to write IDs, where an ID can either be Alpha-Numeric or Numeric only.

All of the above mentioned variations make the job of extracting logical structures extremely challenging.

1.2.1 Requirements for Logical Structure Extraction Systems

The analyses of requirements documents and a survey of related work (see Chapter 3) leads us to determine a set of features necessary for a practical logical structure extraction system. We summarize them in the following points and provide rationale for their inclusion in the requirements for a practical logical extraction system.

1. *The extraction system should work with any type of documents including Word, PDF, HTML etc.*

   Software Requirements may be captured in many different types of documents with a consistent template. HTML and Word documents that conform visually to a single template have radically different underlying Office Open XML [12] for Word and HTML+CSS for HTML documents. Several tools exist that allow conversion from HTML to Word, PDF to Word and vice versa [13][14]. However, the conversion is often erroneous. If the logical structure extraction system defines templates at the level of underlying representation, then separate templates must be defined for different types of documents which is extra work. Therefore, the logical structure extraction system should be able to extract instances from documents of different type given a single template.

2. *The system should allow the templates to be represented in a human-readable form.*

   In case, the templates are induced from a given set of examples, such as in wrapper induction (see Sec.3.2), they should be encoded in a human readable syntax. We

believe that certain parts of the template, for example optional attributes, are expert knowledge and can be edited by the user of the system. For example, a user may already know that preconditions and postconditions in use cases are optional attributes. To completely capture this information using examples will not be practical, as the example set must contain an example where each of these attributes is present and an example where none of these attributes appear. Considering the fact that a logical structure may contain many optional attributes, a completely automated learning tool would require a lot of examples to completely capture the variability in the instances. It is therefore necessary for the template to be encoded in a human-readable form so that a user may add or modify knowledge by changing parts of it. Moreover, these changes should be easy to make.

3. *The system should be able to handle minor errors like spelling mistakes, style inconsistencies etc.*

   Using attribute identifiers such as *Use Case Name* and *Preconditions* in regular expressions to find matching elements will not work if spelling mistakes occur within these identifiers. On the other hand capturing all possible variants of small spelling errors within the regular expression is also not suitable. Therefore, the extraction system should be able to deal with minor errors in identifier parts of the template while still being able to match the pattern specified by the template.

4. *The system should be easily extendable in its capability to recognize and extract new logical structures and presentations.*

   There are many different logical structures used in requirements documents, architecture documents etc. and there are many different ways of presenting them. Logical structures of all kinds such as Functional and Non-functional Requirements, Data Objects, Business Cases and those specific to an organization or project should be extractable.

1.3 Objectives

The objective of this thesis is to provide a comprehensive solution for the extraction of logical structures, such as use cases, functional requirements, business requirements etc. from software requirements documents. The following novel contributions are made in this thesis:

- We identify and present necessary requirements for logical structure extraction systems.

- We present a novel approach to extract logical structures from requirements documents. Our approach uses meta-models of logical structures to extract instances of logical structures.
- Our approach is independent of the type of documents and can be easily extended to extract logical structures from different types (PDF, HTML etc.) of documents.
- Our approach can perform partial matching of the text and style parts of the meta-models to overcome minor editing errors in the instances of logical structures.
- We evaluate our approach on real-world documents collected from various sources including industrial organizations. Our approach achieves high levels of precision and recall and executes within a reasonable amount of time

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides a brief overview of Clafer; the language used by our framework for meta-modeling. Chapter 3 discusses related work and Chapter 4 presents details about different parts of our proposed framework including meta-modeling and the extraction process. We provide and discuss evaluation results in Chapter 5 and conclude the thesis with Chapter 6.

Chapter 2

Background

The meta-models in the proposed framework are specified using Clafer [15]. The following section provides a brief overview of Clafer.

2.1 Clafer

Clafer (class, feature, reference) is a meta-modeling/concept-modeling language. The goal of Clafer is to provide a concise language for class modeling, feature modeling and a mixture of class and feature modeling. Moreover, Clafer can be used to model arbitrary concepts and allows specialization of models by means of constraints and inheritance. Fig. 2.1 shows a simple meta-model for students in a school. Each student in the school has a name, a student id, and a year of graduation. Each student can either be a graduate or an undergraduate student and would be enrolled in one of the degrees (finance, math, economics, cs) offered by the school. A student can take many courses during the course of their degree. Each course has an id and a title. We now explain Clafer using the presented example.

A Clafer model is a set of *type definitions*, *clafers* and *constraints*. The meta-model in Fig. 2.1 has three top-level type definitions Person (line 1), Student (line 7) and Course (line 4). Each type definition contains a hierarchy of clafers (lines 2-3, 5-6, 8-20). Student additionally contains a constraint (line 21). The hierarchy is defined by the indentation of clafers. Each type definition provides a separate namespace for the clafers it contains. An abstract modifier indicates that no instance of the type can be created unless extended by a concrete type.

```
1  abstract Person
2    name : String
3    age : Integer

4  abstract Course
5    courseID : Integer
6    courseTitle : String

7  abstract Student : Person
8    studentID : Integer
9    gradYear : Integer
10   courses -> Course 2..*
11   enrolled ?

12   Info
13     xor Program
14       Grad
15       Undergrad

16     xor Major
17       Finance
18       Math
19       Economics
20       CS
21   [gradYear < 2011 => ~enrolled]
```

Figure 2.1: Meta-model in Clafer

A Clafers can be thought of a slot that can contain instances or references to instances of clafers. Reference clafers are declared using -> and have no sub-clafers. In our meta-model, courses (line 10) is a reference clafer that can contain references to instances of type Course. A Clafer that is not declared using -> such as studentID (line 8), or contains sub-clafers such as Major (line 16), is called a containment clafer i.e. a clafer that contains instances. A containment clafer definition creates a clafer and, implicitly, a new concrete type, both located in the same namespace.

Clafers have clafer cardinality associated with them which puts constraints on the number of clafer instances or references that a given clafer may contain. Cardinality of the clafer is specified as an interval between $m..n$, where $m \in \mathbb{N}, n \in \mathbb{N} \cup \{*\}, m \leq n$. By default, each clafer has an associated cardinality of 1..1. The cardinality 0..1 can be written as "?". The cardinality in line 10 specifies that a valid instance of Student must have a reference clafer courses that contains references to at least 2 instances of Course. Similarly, the "?" in line 11 states that an instance of Student may or may not have an instance of enrolled. Clafers also have an associated group cardinality, which constraints the number of child instances, i.e. the instances contained by sub-clafers. The group cardinality also ranges between $m..n$. **xor** is equivalent to a group cardinality of [1..1]. For example, xor on Major (line 16) in our meta-model states that only one instance of either Finance, Math, Economics or CS is allowed. The xor is equivalent to saying that a student's major can only be one of the above options.

A constraint is used to express dependencies between one or more clafers (line 21, Fig.2.1) or to restrict integer and string values (line 2-3, Fig.2.2). The constraint in line 21, Fig.2.1 states that if a student's graduation year is less than 2011 then the instance of Student must not contain an instance of the sub-clafer *enrolled*. In other words, the constraint states that a student whose graduating batch is less than 2011 is not enrolled.

Clafers can be extended by means of inheritance. Student is inherited from Person and inherits all the clafers of the type Person.

```
1  course1 : Course
2    [courseID = 1
3    courseTitle = "Intro to Algorithms"]

4  course2 : Course
5    [courseID = 2
6    courseTitle = "Information Retrieval"]

7  Rehan : Student
8    [name = "Rehan Rauf"
9    age = 25
10   studentID = 1244342
11   gradYear = 2011
12   courses = course1 + course2
13   Grad && CS]
```

Figure 2.2: Instance of Meta-model in Fig.2.1

The meta-model in Fig.2.1 can have many instances. Each instance is obtained by extending or specializing the meta-model. Fig.2.2 shows an example instance of the meta-model in Fig.2.1. A valid student must be listed in two courses. In our example, two concrete instances of Course (line 1,4) are created by extending from Course. A concrete Student Rehan extends Student and constrains its sub-clafers. The courses (line 12) reference is constrained to point to the two instances of Course. The constraint Grad && CS (line 13) automatically selects the Program and Major clafers. More details about the language can be found elsewhere [15]

Chapter 3

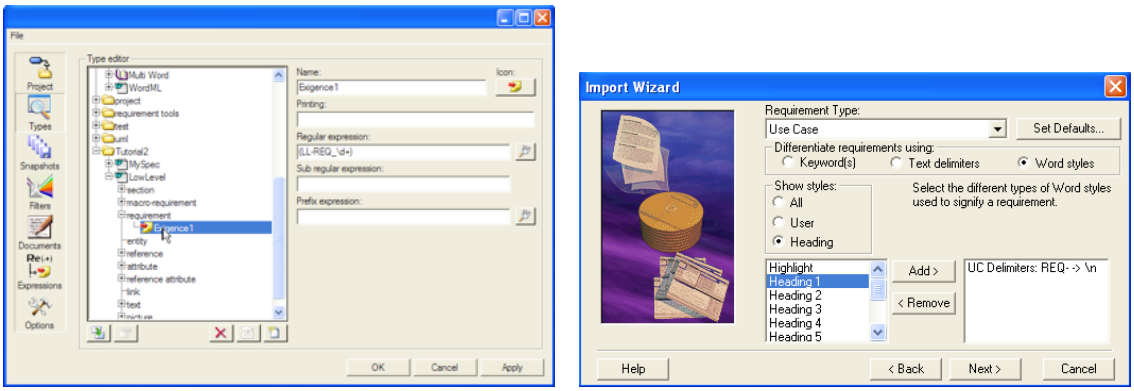
Related Work

We find many examples of automated natural language analysis being used to reason about the quality of requirements, to disambiguate confusing statements and extract ontology [16, 17, 18, 5, 19]. Most of these techniques work at the level of sentences without the knowledge of semantic structures to which the sentences belong. Recently, systems are emerging that perform automatic inspection at the level of logical structures like use cases. Text2Test [20] is one such system that uses text analysis to produce a semantic model for the use case and report problems in the model at edit time. The use cases are either written in the editor provided by the system or are imported from other documents. Working at the level of logical structures opens up new possibilities for a deeper inspection.

As far as we know, there have been no formal studies dealing with extraction of logical structures from software requirements documents. However, some commercially available requirements management tools allow users to extract requirements from documents by means of import. We now present the features and shortfalls of the logical structure extraction provided by two well-known requirements management tools.

3.1 Logical Structure Import in Requirements Management Tools

Reqtify [21] and Requisite Pro[3] are powerful requirements management and traceability tools that allow users to import Word and Excel documents and provide traceability links



(a) Reqtify (b) Requisite Pro

Figure 3.1: Importing Requirements in RM Tools

to elements within the documents. As shown in Fig.3.1[1], these tools allow users to specify a regular expression or a prefix expression to match the elements of interest and import requirements. Requisite Pro additionally allows users to match elements by a pre-defined style (Heading 1). The use of such expressions can only be practical if the authors of the document strictly conform to style and patterns recognized by these tools. For example, a regular expression "REQ-\d+" would match each definition of the requirement of this form as well as each reference to a requirement (Fig.3.2).

REQ-1: Requirement Name 1
System: related system
Description: some description text

REQ-21: Requirement Name 21
System: related system
Description: some description text

(a) Requirement Definitions

REQ-21 is related to ..... some text

(b) Reference to Requirement

Figure 3.2: Definition vs Reference

These tools rely on using predefined Word styles (Headings) to disambiguate between the definition and the reference. It forces the authors to strictly format the document so that it may be imported. This requirement can be enforced on new documents being written but old documents that were not written with this strict requirement in mind will not be importable. Moreover, these regular expressions are good to match individual

[1]Fig.3.1a captured from `http://www.geensys.com/commun/docs/reqtify/Viewlet/tutorial2.viewlet/tutorial2_viewlet_swf.html`

requirements in the form of a paragraph or a section but recognizing attributes within these requirements is not so trivial. Although, theoretically one can define sub-expressions for each attribute, they need to incorporate a lot of variations like missing or out of order attributes. There is also no direct support for requirements structured within tables. These regular expressions cannot be used to extract information from physical structures within tables (columns, rows etc.) without incorporating underlying mark-up tags used for tables. Such application of regular expressions for locating elements is widely found in the field of information retrieval. Next we discuss these approaches briefly and list down their limitations.

3.2 Wrappers

In IR literature, the procedure for the extraction of relational content from webpages is called *Wrapper* [22]. Wrappers are used to crawl webpages and extract specific information from them. These wrappers use known patterns to locate and extract the required knowledge. The process of automatically generating a wrapper based on examples is called *Wrapper Induction* [23]. With the rapid expansion of the Internet and growing hunger for information, many wrapper induction systems have emerged to facilitate extraction of interesting information from webpages [22] [24] [25] [26] [27]. Many toolkits for generating wrappers have also been developed [28].

WIEN[23] is one of the earliest wrapper induction systems. It assumes information to be present in a tabular structure. There are six proposed types of wrappers. The simplest is the Left-Right Wrapper. An example of the LR wrapper would be <B> </B> where <B> is the left delimiter and </B> is the right delimiter. Given many examples, the system is able to generalize the left and right delimiters. Using these delimiters, the system will extract all the elements present between them. All the other proposed wrappers also use delimiters. The proposed Head-Left-Right-Tail wrapper works exactly the same as LR wrapper except that it ignores the Head and Tail of the page which might be using the same delimiters. Such use of delimiters only works with consistently and uniquely tagged information.

A different wrapper induction system WHISK [24] produces wrappers in the form of regular expressions. Given a set of labeled examples and the number of slots that need to be extracted WHISK specializes a regular expression that covers all given examples and extracts information present in the defined slots. For example, if three elements are to be extracted WHISK will start with the basic rule * (*) * (*) * (*) *. The elements that need to be extracted are represented using brackets. These element slots can also specify classes

that need to be matched. Using the labeled examples the rule is specialized. For example a rule :

'.' (NUMBER) * (COUNTRY) '@newline'

when operates on data

No. 1 China
No. 2 India
No.-3 Russia

will extract the tuples (1,china), (2,india), (3,russia). The good this is that WHISK rules also take into consideration the class of the element to extract along with the delimiters. This helps improve precision by eliminating text the fits the pattern but does not belong to the desired class. The problem with WHISK is that it does not allow missing elements nor does it allow different permutations of the elements.

In SoftMealy [29], extraction rules are expressed as Finite State Transducers. The main advantages are that it allows missing element values in generated wrappers and can cater for different permutations. It also allows disjunction in the rules. An example rule :

*. (Number) '@space' Either * ',' (Country) '@newline'
OR (Country) '@newline'

when operates on data

No. 1 China
No. 2 Mumbai, India
No. 3 Russia

will extract the tuples (1,china), (2,india), (3,russia) . One critical problem with SoftMealy induction system is that all the possible permutations of the desired elements must be depicted in the examples. The elements that can be missing must also be indicated by the training examples. Similar to SoftMealy, Borker et al. use Hidden Markov Models to capture the variability in the order and cardinality of attributes.[30] Their proposed technique was shown to work well with human written address records and bibliography records that had variability in the order and number of attributes. However, a separate training set of sufficient size had to be supplied for each type of record. In case of requirements documents, providing sufficient training examples may not be feasible for practical purposes because each organization employs a different template with different attributes, some of which are custom defined for the company.

In [26] the authors propose wrapper induction and extraction techniques for records whose attributes may be present in different branches of the HTML DOM. They use a *broom*

structure which represents a path of HTML tags to represent their records and wrappers. During extraction the path of the *broom* is used to limit the search to portions of the DOM matching the path. Records are extracted by matching the tree part of the *broom*. The technique also makes use of attribute identifiers to disambiguate attributes which appear in similar HTML tags. Although the authors acknowledge that the disambiguation is necessary in the case of optional attributes, they fail to present clear heuristics to do so.

STALKER[25] is one of the more sophisticated wrapper induction systems. In STALKER, rules are represented in a hierarchical fashion using *SkipTo()*. The hierarchy is defined as an Embedded Catalog Tree (ECT), which is similar to the hierarchical model of the document. For example, for an ECT defined as

Document := Some -Text List(UseCases)
UseCase := Name List(Main Scenario)
Main Scenario := List(Scenarios)

and given a document with two example use cases

```
<body>
    some other artifacts
    <UseCases>
        <Table>
            <tr><b>Name:</b> First UC </tr>
            <tr><b>Main Scenario:</b> Scenario1 \n Scenario2 \n Scenario3 </tr>
        </Table>
        <Table>
            <tr><i>Name:</i> Second UC </tr>
            <tr><b>Main Scenario:</b> Scenario4 \n Scenario5 \n Scenario6 </tr>
        </Table>
    </UseCases>
</body>
```

STALKER will first extract the list of use cases then extract each use case and then its *name* followed by the *main scenario*. The extraction of each attribute is guided by the associated rules of extraction. The rules of extraction in the above example are:

Extraction of List (UseCases)	: *‘<UseCases>’(*)‘</UseCases>’
Iteration Rule for UseCases	: *‘<table>’ (*) ‘</Table>’
Extraction Rule for Name	: *‘Name:’ (*) ‘</tr>’ OR *‘Name:</i>’ (*) ‘</tr>’
Extraction Rule for List(Main Scenario)	: *‘Main Scenario:’ (*) ‘<tr>’
Iteration Rule for Scenarios	: ‘@begin’ OR ‘@newline’ (*) ‘@newline’

With STALKER one can extract each logical structure with its attributes. However, all logical and physical variations must be encoded using rules similar to those shown above.

Logical variations can be incorporated by providing different examples but accidental physical variations are very hard to encode in such a manner.

To summarize, Wrappers, which are sophisticated regular expressions, work well in the domain of webpages. However, there are many factors limiting their practical use for the extraction of logical structures from requirements documents. First, the wrappers work with the underlying representation, such as HTML tags, of documents. In case of websites, the pages are usually generated by a script that fills in data from a relational back-end storage. Therefore, the HTML generated is the same for every instance of a particular logical structure. In case of requirements documents in Word, the underlying representation is Office Open XML [12]. While writing logical structures, the users try to visually match the template. This often results in a logical structure that visually conforms to the template but the underlying OOXML is very different from the original template. Therefore a wrapper trained on a template may not be capable of extracting all of the instances. Second, the wrapper induction systems are limited in their capability to efficiently handle missing and out of order attributes and require a large training set to capture all possible scenarios. A way around this problem is to directly change the induced wrapper to take into account missing or out of order attributes. However, in most of the wrapper induction systems we encountered, induced wrappers use system specific and technical representation which is not easy to read for humans and cannot be edited by the users.

3.3 Framework-specific Modeling Languages

The approach presented in this thesis is inspired by the use of framework-specific modeling languages (FSMLs) to represent correct usages of different object-oriented frameworks [31]. Object-oriented frameworks allow developers to instantiate framework concepts by writing framework completion code. A framework-specific model (FSM) represents the implementation code of the application from the framework viewpoint. It provides information about how framework concepts are implemented in the code.

In object oriented frameworks, the instantiation of framework concepts is governed by the application programming interface (API) of the framework. The API exposes the programming elements and their associated rules of use to the developers, for example, what classes to subclass, methods to call and how they should be used. The APIs often provide several different mechanisms to implement a similar concept, for example an eclipse [32] editor can be implemented as single-page or multipage, can register different types of UI event listeners, and can itself be a source of events.

Antkiewicz et al. propose that APIs and their rules actually form a domain-specific language whose concepts are implemented and exposed through the mechanisms of the framework's programming language [31]. The domain-specific language that formalizes the framework's API concepts and constraints is called a Framework-Specific Modeling Language (FSML). FSMLs are used to express API usages by creating a model. Each FSML is developed by an expert familiar with a given framework. The meta-model of the FSML is specified declaratively using cardinality-based feature models [33] and specifies the concepts of the framework and the variability in how they can be used. Each concept and the property of the concept is represented as a feature in the meta-model of an FSML. A feature in an FSML usually has an associated mapping definition, which specifies how the feature is implemented in the code. Mapping definitions are created by providing values to several parameters, such as method signatures or super type names. These parameters are defined in generic and reusable mapping types, such as "a method call to a method with signature <s> in the control flow of a class <c>", in which <s> and <c> are the parameters. From such a mapping type, a concrete mapping definition for a feature can be created by providing concrete values to these parameters. For example, value of s could be given as "Applet.getParameter(String)".

```
1  Applet <class>
2      [1..1] name (String) <fullyQualifiedName>
3     ![1..1] extendsApplet <extendsClass: 'Applet'>
4        [0..1] extendsJApplet <extendsClass: 'JApplet'>
5      [0..*] parameter <callsReceived: 'String Applet.getParameter(String)' >
6        [0..*] name (String) <valueOfArg: 1>
7      [0..*] providesParameterInfo <methods: 'String [ ] [ ] getParameterInfo()'>
8      [1..1] overridesRequiredMethods
9        <1-3>
10       [0..1] init <methods: 'void init()'>
11       [0..1] start <methods: 'void start()'>
12       [0..1] paint <methods: 'void paint(Graphics)'>
```

Figure 3.3: Applet FSML

Fig. 3.3 shows a fragment of the Java AppletFSML [31]. "An applet is a program written in the Java programming language that can be included in an HTML page" [34]. The concepts in the Applet framework are shown as a hierarchy of features. Features with an "!" are called essential features. Essential features imply the parent feature. Mandatory features for example, *name* (line 2) are identified with the cardinality of [1..1]. In Fig. 3.3, the *name* (line 2) feature represents the fully qualified name of the Java class for an applet. *extendsApplet* (line 3) specifies that the Applet class extends the java.applet.Applet. *extendsJApplet* (line 4) specifies whether or not the applet extends java.swing.JApplet. This

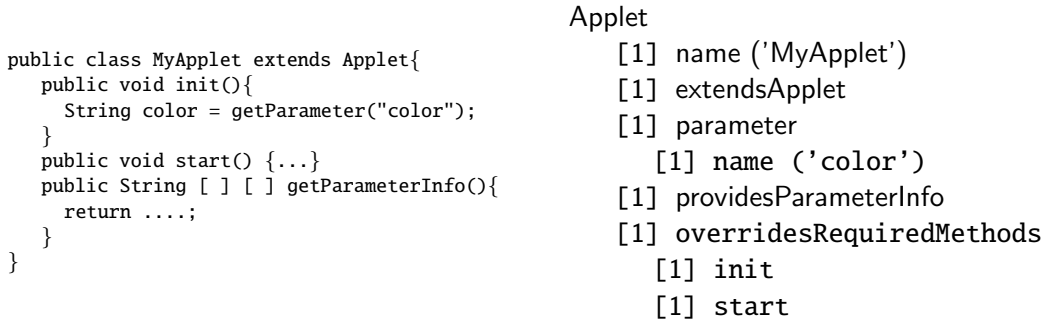


Figure 3.4: Example Applet Framework-specific Model

is depicted by the cardinality for extendsJApplet [0..1].

Lines 5-6 in the Applet FSML represent HTML parameters. Applets are passed different parameters by HTML. These parameters are accessed by making a call to Applet.getParameter(String) and providing a name. The *name* in line 6 represents the name of the parameter that will be passed to Applet.getParameter(String) (line 5). The *providesParameterInfo* (line 7) feature depicts whether the getParameterInfo() method has been overridden to provide parameter information.

Group cardinality is shown as <m-n>. The group cardinality specifies the constraint on the number of instances of children a given feature can have. The *overridesRequiredMethods* (line 8-9) group specifies the rule that the Applet should override at least one of the methods– *init, start, paint.* These methods are shown as sub-features. The associated mapping definition of features is shown using < > brackets. The mapping definition specifies the parameter name, for example, *callsReceived* (line 5), *methods* (line 10) and the parameter value, for example, *String getParameter(String)* (line 5), *void init()* (line 10).

Framework specific models can be extracted automatically by reverse engineering the applications that use a given framework.[35] The application code is statically analyzed to match known patterns for features in the FSML meta-model. The existence of these matches determines the presence of features and associated values. Fig. 3.4 shows an example Applet code and the corresponding Framework-specific model. The framework-specific model depicts the configuration of Applet FSML meta-model for the given code.

The work in this thesis applies similar ideas in the domain of software requirements documents. We hypothesize that a meta-model (equivalent to FSML meta-model) can be defined for each type of logical structure belonging to a certain group of documents. The meta-model can also express variability, constraints on the attributes of the logical

structures, and mapping definitions using concepts similar to the ones used in framework-specific modelling languages. This meta-model can then be used to locate and extract the logical structures. We present the details of our approach in the next chapter.

Chapter 4

Framework for Logical Structure Extraction

We propose a framework for model-guided extraction of logical structures. The framework consists of three parts – a modeling part, document queries and an extraction tool, ET. In the framework, a meta-model is created for each logical structure that is to be extracted. The meta-model specifies various attributes a logical structure may have as well as cardinality (optional, exactly one or many) of attribute instances and variations in attribute identifiers. The meta-model also captures the information about how the logical structure is materialized in the document. The extraction tool, ET, consumes a meta-model and uses document queries to extract elements from the documents. It evaluates the extracted elements against constraints specified by the meta-model and produces only the satisfying elements as instances of the meta-model.

4.1 Meta-Modeling

The framework uses Clafer (see Sec.2.1) to specify the meta-models. The language allows specialization of models by means of inheritance and constraints. The meta-model in the framework consists of three basic types: *LogicalStructure*, *Attribute* and *Mapping*. The type *LogicalStructure* is the most abstract type for any logical structure. All logical structures extend from the type LogicalStructure. Each logical structure is composed of many attributes. Attributes extend from the type *Attribute*. Physical presentations are modeled as Mappings. We observe that each document is made up of three primitive building blocks; paragraph, cell (table cell) and a graphic object. All other elements are

built using these basic building blocks. For example, a list is a collection of enumerated or bulleted paragraphs and a column in a table is an arrangement of cells in a specific order. We represent each basic element by a Mapping in our framework. Complex Mappings like list, section, column in a table are composed of basic Mappings like ParagraphMapping, CellMapping etc. The reason these presentations are called Mappings is because they *map* an abstract meta-model to materialized instances in the documents.

4.1.1 Mappings

In the meta-model, a mapping is specified with the logical structure and each of its attributes. The mapping provides information about the physical element in the document that the logical structure or attribute maps to. To identify mappings, we chose a random sample of 20 software requirements documents from our collection and manually inspected the logical structures and attributes present in them. We identified a set of 15 physical presentations that were commonly used throughout these documents to author the logical structures and their attributes. We analyzed the properties of these physical presentations and came up with the a set of 15 mappings with multiple parameters. The mappings are are also defined using Clafer.

```
abstract SectionMapping : Mapping
    sectionTitleText:String?
    sectionTitlePattern:String?
    sectionTitleStyle->LSStyle?
```

(a) Definition

```
'SectionMapping
    [sectionTitleText = "Requirements"]
    [sectionTitleStyle = Bold12]
```

(b) Example

Figure 4.1: SectionMapping

Each mapping extends from the type *Mapping* and includes parameters to specify properties of the mapping. For example, the SectionMapping, shown in Fig. 4.1a, consists of three parameters: sectionTitlePattern, sectionTitleText and sectionTitleStyle. The "?"at the end specifies that the parameter is optional and may or may not be specified. The parameters of SectionMapping can be constrained to represent a set of particular sections in the documents. For example, the SectionMapping defined in Fig. 4.1b is constrained to represent only the sections whose title is "Requirements"and the title has a specific style i.e. Bold12. In Clafer, 'SectionMapping is equivalent to SectionMapping:SectionMapping which defines a concrete clafer (SectionMapping) that has the same name as the type (SectionMapping) it extends from.

```
‘SectionMapping
    [sectionTitlePattern = “Functional {Req-‘NUM’}:*"]
    [sectionTitleStyle = Bold12]
```

Figure 4.2: SectionMapping with Title Pattern

If the section title is not consistent across modelled sections, sectionTitlePattern can be specified as shown in Fig. 4.2. The convention used to express patterns in the framework is similar to the one used by WHISK [24]. The patterns are composed of three regular expressions and are of the form “regex {regex} regex”. The whole expression defines the pattern that needs to be matched and the expression within the parenthesis specify the portion to which the attribute maps to. Special reserved symbols, such as ‘NUM’are defined to represent common regular expressions. They are used within (‘’) quotes in the pattern expressions. For example, the reserved symbol ‘NUM’is equivalent to a regular expression that will match numbers including those containing decimal points. This makes it easier for the user to read the meta-model.

We believe that few mappings should be enough to model many different logical structures. However, if required, new mappings can be defined. The 15 mappings that were identified were enough for modelling all the logical structures encountered in the set of evaluated documents. These mappings are presented in Appendix A.

```
abstract LSStyle
    bold?
    italic?
    fontSize->Integer
```

(a) Definition

```
Bold12:LSStyle
    [bold]
    [~italic]
    [fontSize = 12]
```

(b) Example

Figure 4.3: Style

4.1.2 Style

The style in a meta-model is represented by three properties – bold, italic and font size. Different styles are represented by extending the basic type LSStyle which has three parameters bold, italic, fontSize. The bold and italic parameters are boolean parameters where as the fontSize is an integer parameter. An example Style object is shown in Fig. 4.3b. Semantically, it means that the element with which this style may be associated to is bold, not italic and has a font size of 12.

4.1.3 Meta-Models

We explain the meta-models in our framework by providing examples. The meta-model for use cases in Fig.1.1a and Fig.1.1b are shown in Fig.4.4 and Fig.4.5 respectively. We now explain in detail the meta-model in Fig.4.4. Semantically the meta-model reads: Instances of logical structure UseCase1 map to a section whose title is bold and the font size of the title is 12. UseCase1 has four attributes ID, Name, Flow and Extensions. Extensions may or may not be present in some instances of UseCase1. ID and Name map to the title of section the use case maps to. The portion of section title that belongs to the ID and Name is defined by their respective patterns. Flow maps to a list and Extensions map to either a sub-section with title "Extensions"or a text block within the section the use case maps to.

```
1  abstract UseCase1 : LogicalStructure
2    'SectionMapping
3      [sectionTitleStyle = Bold12]

4    ID : Attribute
5      'SectionTitleMapping
6        [sectionTitlePattern="{UC 'NUM'} * ]"

7    Name : Attribute
8      'SectionTitleMapping
9        [sectionTitlePattern=" UC 'NUM' {*} ]"

10   Flow : Attribute
11     'ListMapping
12     FlowItem : Attribute 1..*
13       'ParagraphMapping

14   Extensions : Attribute ?
15     xor Mapping
16       'SectionMapping
17         [sectionTitleText="Extensions:"]
18       'TextBlockMapping
19         [identText="Extension"]
20         [delimiter=":"]
```

Figure 4.4: Meta-model for Use Cases in Fig.1.1a

Line 1 in the meta-model defines the logical structure to be extracted. Line 2 creates a *clafer* named SectionMapping of type SectionMapping in the scope of UseCase1. This SectionMapping defines the mapping for the logical structure. Line 3 constraints the

sectionTitleStyle parameter of the SectionMapping clafer to refer to the clafer Bold12 that is defined outside the scope of UseCase1. Lines 4-20 create ID, Name, Flow and Extensions clafers of super type Attribute in the scope of UseCase1. Each attribute contains a mapping defined within its scope.

```
1  abstract UseCase2 : LogicalStructure
2    'TableMapping

3    Name : Attribute
4      'CellMapping
5        [colIndex=1
6         rowIndex=1]

7    Actor : Attribute
8      'HCellBlockMapping
9        [identText="Actor"]

10   Precondition : Attribute ?
11     'HCellBlockMapping
12       [identText="Precondition"]

13   xor Flow
14     ProcDesc : Attribute
15       'ColumnMapping
16         [colTitleText = "Process Description"]
17       Action : Attribute
18         'ColumnMapping
19           [colTitleText = "Action"]
20       Response : Attribute
21         'ColumnMapping
22           [colTitleText = "Response"]
23     MainScenario : Attribute
24          'ColumnMapping
25            [colTitleText = "Main Scenario"]
```

Figure 4.5: Meta-model for Use Cases in Fig.1.1b

An attribute may consist of many sub-attributes that can be identified individually. For example, the Flow in line 10 of Fig. 4.4 maps to a list. Each item of the list can be recognized by defining a sub-attribute FlowItem in the scope of Flow. Line 12-13 show the declaration of the sub-attribute FlowItem. Similarly, lines 14-22 in Fig. 4.5 show Process Description with two sub-attributes – Action (Line 17) and Response (Line 20).

Logical Variations

Logical variations in the logical structures can be modeled within the meta-model. For example, use cases in Fig.1.1b use either the "Main Scenario" or the "ProcessDescription" attributes to capture the Flow of a use case. This is modelled as an **xor** group in the meta-model.

```
13 xor Flow
14   ProcDesc : Attribute
23   MainScenario : Attribute
```

Logical Structures may have attributes that are optional such as preconditions or attributes that may have different number of instances within one logical structure, for example, actor in a use case. The cardinality of attributes can be specified easily by specifying the cardinality range with the definition of the attribute. The default cardinality range for each defined attribute is [1..1], which means at least one and at most one. The cardinality range for an optional attribute is [0..1] and can be denoted by "**?**". FlowItem in Fig.4.4 (line 12) has a cardinality range of [1..*] which means that there should be at least one flow item but there can be many flow items. Similarly, "**?**" in line 14, Fig.4.4 and line 10, Fig.4.5 specifies that Extensions and Preconditions are optional.

Designed Physical Variations

Designed variations in physical presentation are modeled by having an **xor** group for the mappings. For example, the meta-model in Fig. 4.4 (lines 15-20) allows *Extensions* to be written either as a section or a text block.

```
15 xor Mapping
16   'SectionMapping
18   'TextBlockMapping
```

Accidental Physical Variations

To overcome accidental variations, a threshold is used for both: text matching and style matching. The thresholds range from 1 to 100 and can be specified in the meta-model itself by including the ExtractionSettings. Fig.4.6 shows an example threshold setting. Setting the threshold to 100 means that an exact match will be required between the text and style

identifiers specified by the meta-model and the text and style of the document elements being matched.

```
ExtractionSettings
    [textMatchThreshold = 90]
    [styleMatchThreshold = 70]
```

Figure 4.6: Example Extraction Settings

Scope for Logical Structures

Sometimes it may not be necessary to look for logical structures in all parts of the documents. For example, if we find that non-functional requirements are always written in the appendix section of the requirements documents, we can limit the scope of our search by providing an optional parameter lsScope for Logical Structures. Fig.4.7 shows a portion of the meta-model for such a scenario.

```
1 abstract NonFuncReq : LogicalStructure
2   'TableMapping
3   'SectionMapping
4       [sectionTitlePattern = "*Appendix*"]
5   [lsScope = SectionMapping"]
```

Figure 4.7: Scope for Logical Structures

Lines 2-4 create two Mapping clafers – TableMapping and SectionMapping. In line 5, the SectionMapping clafer is assigned to the lsScope parameter of the logical structure NonFuncReq. TableMapping is automatically considered to be the mapping for logical structure NonFuncReq. Semantically, the meta-model reads: Non functional requirements map to a Table in the Section whose title contains the word "Appendix". For the given meta-model, tables found only in appendices of the documents will be included in the search for non functional requirements. Defining lsScope may also be required when the only thing that uniquely distinguishes between two separate types of logical structures is the context in which their instances are found.

```
1 abstract BusinessRule : LogicalStructure
2   'SectionMapping
3   [ordered]
4   ........
```

Figure 4.8: Order of Attributes in Logical Structures

Order of Attributes

By default, the order of attributes is not important in the meta-model. The extraction process looks for attributes with in the scope of logical structures without the restriction on order. However, if the order of the attributes is important to uniquely identify the given logical structure from other structures in the documents, the ordering restriction can be put by specifying an optional parameter ordered? of the logical structure. Fig.4.8 shows a fragment of a meta-model for Business Rules. The ordering restriction is specified by writing a constraint (line 3).

CollectionMapping

Some logical structures may not map to a single define-able element in a document but map to a collection of elements in the document. Fig.4.9 shows use case instances that are written using a combination of text-blocks and a table only. There is no single element that the use case maps to. To model such a scenario, a special Mapping is used i.e. CollectionMapping. Fig.4.9 also shows the meta-model for these use cases. The extraction tool handles the CollectionMapping in a special manner. We explain that in section 4.3.1.

4.2 Document Queries

All mappings have corresponding document queries that are used to extract the document elements modelled by the mapping. For example, the mapping *ListMapping* has a corresponding document query, which can extract lists in the documents that satisfy the parameters of the query. Each Mapping that is provided in the framework must have an implementation of a corresponding document query and should have support for all the parameters of the Mapping. We implemented document queries for Microsoft Word for the 15 mappings identified in Sec.4.1.1.

ID: UC 1
Name: Select Product Category

Action	Response
User selects the option to display all product categories	System displays the list of product categories
User selects one product category	System shows the list of products in the selected product category

ID: UC 2
Name: Add Product to Cart

Action	Response
User chooses a product to add to cart	
User confirms the addition of product to cart	System adds the selected product to cart.

```
abstract UseCase3: LogStruct
  'CollectionMapping

  ID : Attribute
    'TextBlockMapping
      [identText="ID"]

  Name : Attribute
    'TextBlockMapping
      [identText="Name"]

  Description : Attribute
    'TableMapping
```

Figure 4.9: Example of CollectionMapping

Before implementing queries, elements (paragraphs, lists, sections, tables etc.) in the documents need to be identified. There are two types of document elements – *Basic elements* like paragraph, cell and graphics and *Complex elements* like sections, lists, columns etc. We now explain how these two different types of elements are identified.

4.2.1 Basic Elements

Basic elements can be identified directly. In case of MS Word, paragraphs, tables and graphics can be extracted directly by either using the Word API or by parsing the underlying OOXML. We chose to extract individual cells rather than tables. This allowed us to define more complex mappings over a collection of cells than was possible over a collection of tables. Table 1 shows a summary of basic elements and their properties that were identified.

	Properties	Type	Value
Paragraph	text	String	text of the paragraph
	bold	Int	0 if the entire paragraph is not bold. 1 if some parts of paragraph are bold. 2 if the entire paragraph is bold
	italic	Int	0 if the entire paragraph is not italic. 1 if some parts of paragraph are italic. 2 if the entire paragraph is italic
	underline	Int	0 if the entire paragraph is not underlined. 1 if some parts of paragraph are underlined. 2 if the entire paragraph is underlined
	fontSize	Int	Font Size of the paragraph text
	listIndicator	Boolean	True if paragraph belongs to any list. False if it cannot be determined
	rangeStart	Int	Start position of the paragraph relative to the start of the document
	rangeEnd	Int	End position of the paragraph relative to the start of the document
	indent	Int	left-indentation of paragraph
Cell	tableID	Int	ID of the table the cell belongs to
	coordinates	Int	(x_1, y_1) and (x_2, y_2) coordinates of the cell
	rangeStart	Int	Start position of the contents of the cell relative to the start of the document
	rangeEnd	Int	End position of the contents of the cell relative to the start of the document
Graphics	ID	String	ID of the graphics.
	caption	String	Text of caption on the graphic
	rangeStart	Int	Start position of the graphics object relative to the start of the document
	rangeEnd	Int	End position of the graphics object relative to the start of the document

These properties of basic elements are used by the document queries to extract relevant elements. They are also used to identify the complex document elements.

4.2.2 Complex Elements

Complex elements such as Sections, Rows and Columns in a Table, Lists etc. cannot be directly extracted. These complex elements are composed of basic elements. For example, a section is an arrangement of paragraphs, tables and graphics in a specific hierarchy. The heuristics for the identification of complex elements were developed by analyzing the the same set of 20 documents that was used to identify Mappings (Sec.4.1.1). Properties of basic elements are used in the heuristics. Here we present heuristics for the identification of two complex elements, Sections and Columns.

Section

A document can be viewed as a tree of sections. Fig.4.10 shows a small document and the corresponding sections tree.

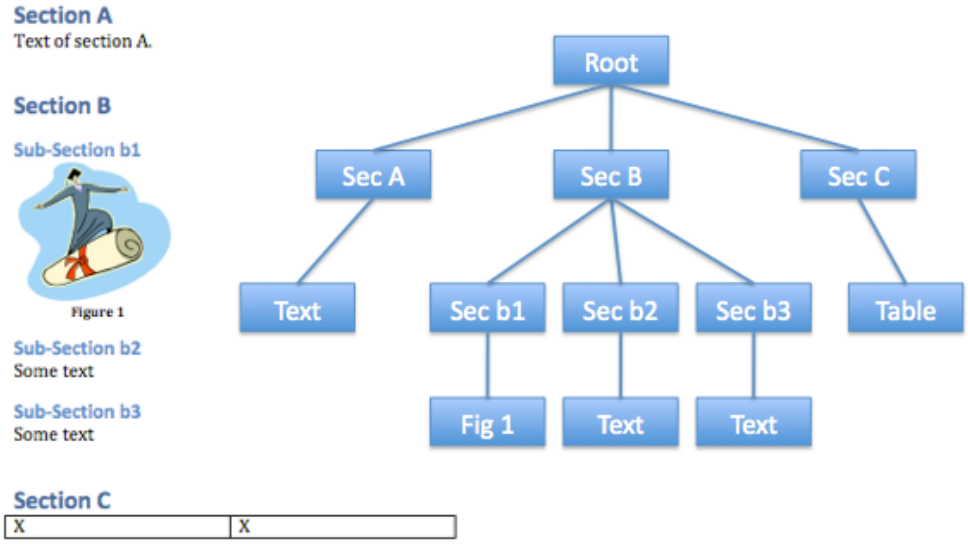


Figure 4.10: Section Tree

The document can be divided into sections by analyzing the style of paragraphs. The following pseudo-code describes the creation of sections tree.

Algorithm 4.2.1: MAKESECTIONSTREE($basicDocElements, root$)

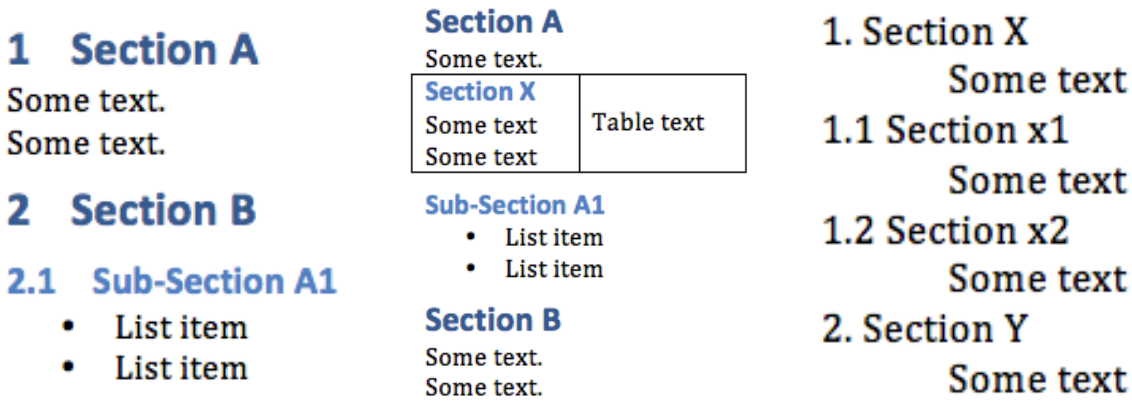
```
prevSection ← root
for each paragraph ∈ basicDocElements
  do { currectSection ← MAKESECTION(paragraph)
       ADDSECTIONTOTREE(currentSection, prevSection)
return (root)

procedure ADDSECTIONTOTREE(currentSection, prevSection)
 cStyle ← currentSection.sectionTitleStyle
 pStyle ← prevSection.sectionTitleStyle
 comparison ← COMPARESTYLES(cStyle, pStyle)
 if comparison > 0
   then { ADDSECTIONTOTREE(currentSection, prevSection.parent)
   else if comparison < 0
   then { prevSection.children ← prevSection.children + currentSection
   else { parent ← prevSection.parent
          parent.Children ← parent.Children + currentSection

procedure COMPARESTYLES(style1, style2)
 if style1.bulletStyle > style2.bulletStyle      return (1)
   else if style1.bulletStyle < style2.bulletStyle      return (−1)
 if style1.fontSize > style2.fontSize      return (1)
   else if style1.fontSize < style2.fontSize      return (−1)
 if style1.bold > style2.bold      return (1)
   else if style1.bold < style2.bold      return (−1)
 if style1.italic > style2.italic      return (1)
   else if style1.italic < style2.italic      return (−1)
 return (0)
```

The comparison of *bulletStyles* involves comparing the bullet numbers of the paragraphs. Fig. 4.11 shows some of the sections that can be identified by the above heuristics. There are 3 sections in Fig. 4.11a i.e. Section A, Section B and Section A1 where Section A1 is a sub-section of Section B. The hierarchy can be determined by comparing the style and bullet number of the section title *paragraphs*. Similarly, Fig. 4.11b shows 4 different sections. As shown, sections can be embedded inside tables as well. In Fig. 4.11b, only

the information about styles is used to determine the hierarchy of sections whereas in Fig. 4.11c, only the information about bullet numbers is used to determine the sections and their hierarchy.

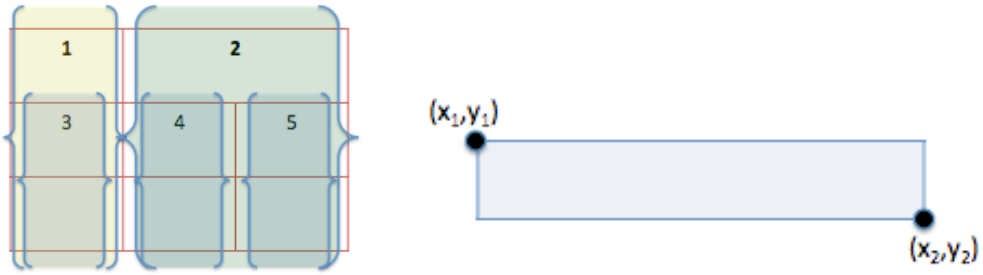


(a) (b) (c)

Figure 4.11: Examples of sections

Column

A column is a vertical arrangement of cells. Fig.4.12a shows all the columns (5) in the given table. The (x1,y1) and (x2,y2) coordinates of the cells are used to determine if the cells are vertically aligned.



(a) Column in a Table (b) Coordinates

Figure 4.12

The following pseudo-code describes the identification of columns.

Algorithm 4.2.2: MAKECOLUMNS(*cellArray*)

```
output ← ⟨⟩
for each cell ∈ cellArray
  do
if cellIsNotEmpty
  then { headCell ← cell
         nextCell ← cell.next
         column ← ⟨⟩
         while true
           do
           { if nextCell.ISVERTICALLYBELOW(headCell)
             and nextCell.OVERLAPSHORIZONTALLYWITH(headCell)
               then { if nextCell.ISWITHINHORIZONTALBOUNDARIESOF(headCell)
                        then { column ← column + nextCell
                               nextCell ← cell.nextCell
                        else break
         output ← output + column
return (output)
```

All other complex elements are identified using similar heuristics. It is worth noting that new document types like PDF, HTML etc. can be incorporated by providing mechanism to extract basic elements. However, the heuristics for complex elements remain the same and hence they need not be re-implemented.

4.2.3 Queries

Each document query executes within a given *scope*. The default scope for each query is *repository*. The scope *repository* specifies that the query will be executed over all the documents in the repository. This means that a simple SectionQuery corresponding to SectionMapping with no parameters will return all the sections in all the documents in the repository. The result of one query can be used as a scope of the next query. For example, each section returned by the SectionQuery can be used as a scope for ListQuery that will return all the lists within the provided section only.

Appendix A presents different queries corresponding to the mappings we identified and the document elements that are matched.

4.2.4 Partial Matching

The extraction tool uses a bi-gram similarity measure [36] to match text parameters in the meta-model such as sectionTitleText and identText with elements in the documents. The bi-gram similarity is computed by breaking down the two strings into bi-grams and computing the similarity coefficient. The coefficient used is the Dice coefficient given by :

$$StringSimilarity = 2\frac{|B \cap B'|}{|B| + |B'|}$$

where B is the set of bi-grams in string 1 and B' is the set of bi-grams in string 2.

The following coefficient is used to match two styles:

$$StyleSimilarity = \frac{(b == b') + (i == i') + \frac{maxF - |f - f'|}{maxF}}{3}$$

where b and b' are bold parameters, i and i' are italic parameters, f and f' are font sizes of the two styles and $maxF$ is the maximum font size. *(b==b')* produces 1 if both bold parameters are the same and 0 otherwise. For our experiments we chose $maxF$ to be 20.

The range for both coefficients is [0,1], which is scaled to [0,100]. One factor that restricts the value of thresholds is the similarity between attribute identifiers used in the meta-model of the logical structures. For example, if a logical structure Use Case consists of two attributes Preconditions and Postconditions both of which map to a text-block and use "preconditions" and "post-conditions" for attribute identifiers, then the text similarity between these two identifiers is 69. This limits the textMatchThreshold which cannot be less than 70 otherwise an instance of attribute *Postcondition* may be identified as an instance of *Precondition*. Selecting the threshold values depends on the parameters used in the meta-model and their role in uniquely identifying the logical structures. If there are many style parameters used without accompanying text parameters then the threshold for style should be set to a higher value to help disambiguate instances of modelled logical structures from other things in the documents. If the threshold is not specified in the meta-model, a default value of 80 is used.The value 80 worked well for most of the documents that were evaluated.

4.3 The Extraction Tool

We implemented an interpreter for logical structure definitions in Clafer. The interpreter is used by the extraction tool to query and reason about the meta-model. To find and extract instances of the meta-models, an extraction tool takes as input, a meta-model, like the one shown in Fig.4.4 and a repository containing project documentation and produces the extracted instances as follows:

1. The algorithm begins by locating all elements in the documents that are referenced by the mapping of the logical structure. In example meta-model Fig.4.4, all sections that conform to the style Bold12 in the documents are located. It is important to note that elements of interest are found using document queries corresponding to each mapping.

2. The algorithm then traverses all the located elements (sections in the example) and searches for attributes (ID, Name etc.) **within the scope** of each element (section). The search for attribute uses the mapping defined for the attribute and its corresponding document query. In the example, ID and Name are searched within the title of each section being traversed. The Flow is mapped to the list element found by the ListQuery. The scope for *ListQuery* is the section being traversed. Two separate document queries – SectionQuery & TextBlockQuery are launched for theExtensions attribute. The Extensions attribute is mapped to either a sub-section or a text-block or nothing based on the results of the queries.

3. Moving forward, the element (section) is checked for constraints on the cardinality of attributes as well as extra constraints specified by the meta-model (for example *order* of the attributes). If all the mandatory attributes are found within the element being traversed and the element satisfies the constraints specified by the meta-model, it is extracted as a specialized instance of the meta-model for that logical structure.

4.3.1 Dealing with CollectionMapping

Due to the nature of CollectionMapping, it has to be treated differently. A possible implementation for the collection query may take as input all the attributes that are defined in the scope of the logical structure and search for ordered occurrences of these attributes. However, to avoid enforcing a complete order, the collection query requires that the first attribute defined must be in order and **not** optional. Based on the first attribute the collection query divides the scope into collections. In the example presented in 4.9, the

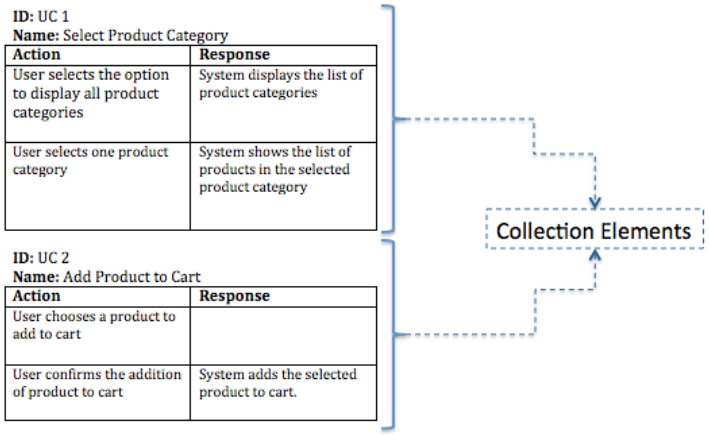


Figure 4.13: Identified Collection Elements in Example 4.9

collection query would search for the attribute ID by using the text-block query and divide and produce collection elements as shown in Fig.4.13. From this point onwards, the extraction process remains the same.

Chapter 5

Evaluation

5.1 Analytical Evaluation

We now discuss the evaluation of the proposed framework in light of the requirements determined in section 1.2.1.

1. *The extraction system should work with any type of documents including Word, PDF, HTML, etc.*

   Logical Structures in the framework are modelled at an abstract level of Sections, Tables, Columns, Lists, etc. The document queries use basic (Paragraph, Cell etc.) and complex (Section, List etc.) document elements in ignorance of the way these elements are identified. The mechanism for the identification of basic elements must be developed for all types (PDF, Word, HTML etc.) of documents. However, as shown in section 4.2.2, the identification of complex elements is based on the properties of the identified basic elements and hence does not require extra work. For the evaluation, the document queries for MS word were created but support for PDF or HTML files can be added easily by developing mechanism for the identification of basic PDF or HTML elements such as *PDFCell, PDFParagraph, HTMLParagraph* etc. This separation of document queries from the structure of elements allows one to specify a single meta-model, which can be used to extract instances from different types of documents.

2. *The system should allow the templates to be specified in a human-readable modelling language.*

The framework uses Clafer to create meta-models. As shown in Fig.4.4 and Fig.4.5, the meta-model in Clafer is easy to read and understand. Moreover, the structure of the meta-model resembles the structures of logical structures i.e. the attributes are defined within the scope of logical structure and attributes that appear within other attributes are recursively defined within in the scope of parent attributes.

Logical and Physical variations can be defined easily. For example, if an attribute has to be made optional, an addition of only a single character "?" is required in the meta-model.

3. *The system should be able to handle minor errors like spelling mistakes, style inconsistencies, etc.*

   Using the two similarity metrics for text and style matches defined in Sec. 4.2.4, minor errors in spellings and style are overcome. The user is given an option to set an acceptable degree of error by means of thresholds.

4. *The system should be easily extendable in its capability to recognize and extract new logical structures and presentations.*

   Many different logical structures such as functional requirements, feature requests, business case etc. can be modelled using the presented approach. The approach only requires that each logical structure be modelled using our meta-modelling specification. The extraction tool is completely decoupled from the type of logical structures. It only requires a valid meta-model. Therefore, the scope of the approach is not limited to software requirements documents only. This approach can be used to model logical structures in rich text documents from any domain.

5.2 Experimental Evaluation

In this section we provide the results of the following investigations:

1. Can we model and extract logical structures from real world software requirements documents?
2. How efficient is the extraction tool?
3. How complex are the meta-models?
4. How do instances of logical structures vary?
5. How does capturing variability affect the complexity of the meta-model?

6. How critical is the need for a human editable meta-model?

5.2.1 Set-up of the Experiments

A total of 46 software requirements documents were collected at the convenience of the authors [37]. These documents belonged to industrial systems, student projects done as part of a course at University of Waterloo, requirements documents collected for the use case database [38] and documents downloaded from the Internet by searching for keywords like "Software Requirements Documents", "SRS", etc. 20 of these documents were randomly selected for the analysis of logical structures and develop the foundations of the framework.

We selected a total of 36 requirements documents in MS Word format from the 46 documents we collected. The other 10 documents were not considered for evaluation because they had very few instances of logical structures. Table 5.1 gives details about the 36 selected documents. The length of each document in terms of total number of words and source of the documents are given in column 2 and 4 respectively. Column 3 of Table 5.1 lists contents of the documents. There are 5 documents (Doc # 21-24, 36) from industry containing only one instance of use case, each encompassing the whole document. The rest of the documents contain multiple instances of logical structures.

Doc #	Length	Content	Source
1	5,239	UC, NFR, Desc	UCDB
2	6,327	DO, UC, NFR, Desc	UCDB
3	3,195	UC, NFR, Desc	UCDB
4	2,387	BC, UC, Desc	UCDB
5	2,104	BC, UC, Desc	UCDB
6	2,932	DO, UC, NFR, Desc	UCDB
7	4,663	UC, NFR, Desc	UCDB
8	12,715	FR, UC, TM, UCD, NFR, GR	Student
9	22,667	FR, UC, TM, UCD, NFR, GR, IR	Student
10	23,003	FR, UC, TM, UCD, NFR, SD, IR	Student
11	20,381	FR, UC, TM, UCD, NFR, GR, IR	Student
12	14,147	FR, UC, TM, UCD, NFR, GR, IR	Student
13	14,154	FR, UC, TM, UCD, NFR, GR, IR	Student
14	12,189	FR, BReq	Industry1
15	2,125	FR, BReq, BR	Industry1
16	1,657	FR, BReq, BR	Industry1
17	2,837	FR, BReq, BR	Industry1
18	2,298	FR, BReq, BR	Industry1
19	4,329	FR, BReq, BR	Industry1
20	2,646	FR, BReq, BR	Industry1

Doc #	Length	Content	Source
21	2,176	UC	Industry2
22	1,319	UC	Industry2
23	5,151	UC	Industry2
24	5,388	UC	Industry2
25	1,566	UC	Internet
26	5,204	UC, Desc, IR, FR, DBR	Internet
27	1,905	GR, UC, Desc	Internet
28	2,441	UC, Desc, Archi Design, Proj Plan	Internet
29	2,909	FR, GR	Internet
30	8,328	UC, NFR, FR, Desc, Risks	Internet
31	3,360	UC, FR	Industry3
32	2,107	UC, Hardware Interface, FR, NFR	Industry3
33	4,741	UC, Hardware Interface, FR, NFR	Industry3
34	2,790	UC, NFR, GR, Desc	Industry3
35	6,757	System Features, Desc, UI	Industry4
Legend	**UC** = Use Case, **FR** = Functional Requirement, **NFR** = Non-functional Requirement, **Desc** = Description of the System, **IR** = Interface Requirement, **GR** = General Requirement, **UI** = User Interface Specification, **BR** = Business Rule, **BReq** = Business Requirement, **UCD** = Use Case Diagram		

Table 5.1: Documents Used in Evaluation

The interpreter for logical structure definitions in Clafer, the extraction tool and document queries are implemented in C#. All experiments were executed on a laptop with a Core Duo 2 @2.26 GHz processor and 4GB of RAM, running Windows.

We evaluated different aspects of the framework by creating meta-models for 33 logical structures. The meta-models were created using an iterative process. For each set of logical structures, we began by taking a few example instances and created a meta-model that conformed to those instances. The meta-model was then fed to the extraction tool. The output of the extraction tool was manually inspected to identify the instances missed by the extraction tool. The meta-model was refined to incorporate the variability that led to the exclusion of those instances from the results. The process was repeated until the maximum recall and precision was obtained for each meta-model. The results of the extraction tool were outputted as an XML file that contained information about all the instances of logical structures and their attributes that were recognized in the provided data set. The XML also had information about the document name and character location of the recognized instances. To help ease the process of manual inspection, the recognized instances were highlighted in the actual documents. To help distinguish between different attributes instances and view their start and end boundaries, varying colors were used for

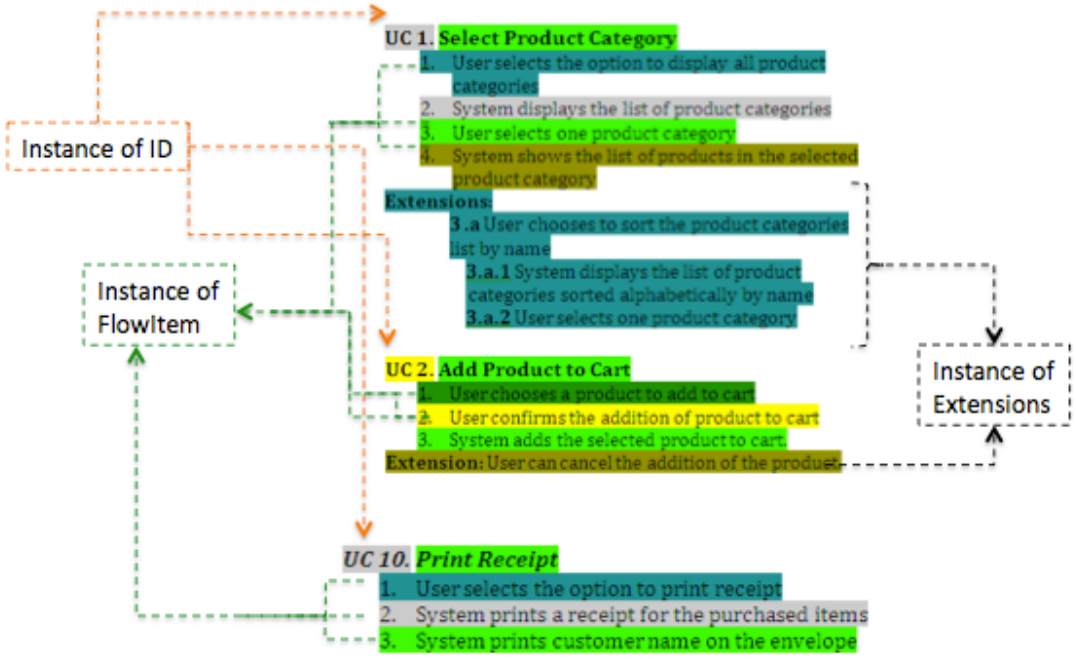


Figure 5.1: Recognized Use Cases in Fig.1.1a

highlighting attributes. Fig. 5.1 shows the highlighted parts after the extraction tool was run on the example use cases in Fig. 1.1a with the meta-model in Fig. 4.4.

5.2.2 Evaluation

We now provide details of the relevant experiments we conducted to investigate the questions outlined above. We present and discuss the results of our experiments.

5.2.3 Can we model and extract logical structures from real world software requirements documents?

As explained above, the meta-models were created in an iterative manner by looking at example instances, creating and refining the meta-model instances until all instances of the meta-model were retrieved accurately or a point was reached where refining the meta-model further was not possible. Given a meta-model for a logical structure with many

attributes, each retrieved item can be either

- an instance of the logical structure with all its attributes correctly recognized
- an instance of the logical structure some of whose attributes are not recognized correctly
- an *item* that is not an instance of the logical structure

The recall [39] is defined as :

$$recall = \frac{|T \cap R|}{|T|}$$

where

T = the set of all the instances of a given logical structure
R = the set of *items* that were retrieved

Precision is defined at two different levels: at the level of the instance of the logical structure ($precision_l$) and at the level of attributes of the logical structure ($precision_a$)

$$precision_l = \frac{|T \cap R|}{|R|}$$

$$precision_a = \frac{|C \cap I|}{|I|}$$

where

T = the set of all the instances of a given logical structure
R = the set of *items* that were retrieved
C = the set of instances with all their attributes recognized correctly
I = the set of retrieved *instances*

The precision and recall for each meta-model for respective logical structure is shown in Table 5.2. Table 5.2 also shows the type of logical structure (column 2) for which the meta-model was created and the documents used (column 3) in the testing set for each meta-model. The information about the content of these documents is shown in Table 5.1.

Meta-Model	**LS**	**Doc #**	**Recall**	**Precision**$_l$	**Precision**$_a$
MM1	UC	1	1	1	1
MM2	NFR	1	1	1	1
MM3	UC	2	1	1	1
MM4	NFR	2	1	0.87	1
MM5	DO	2	1	1	1
MM6	UC	3	1	1	1
MM7	NFR	3	1	1	1
MM8	UC	4	1	1	1
MM9	UC	5	1	1	1
MM10	UC	6	1	1	1
MM11	NFR	6	1	1	1
MM12	DO	6	1	1	1
MM13	UC	7	1	1	1
MM14	UC	1-3,6-7	1	1	1
MM15	UC	21-24,36	1	1	1
MM16	BR	14-20	0.97	1	1
MM17	FR	14-20	1	1	1
MM18	UC	25	1	1	1
MM19	UC	26	1	1	1
MM20	UC	27	1	1	1
MM21	UC	28	1	1	0.86
MM22	FR	29	1	1	1
MM23	UC	30	1	1	1
MM24	UC	8	1	1	1
MM25	UC	9	1	1	1
MM26	UC	10	1	1	1
MM27	UC	11	1	1	1
MM28	UC	12	0.95	1	1
MM29	UC	13	1	1	1
MM30	UC	31	0.83	1	1
MM31	UC	32-33	1	1	1
MM32	UC	34	1	1	1
MM33	SF	35	1	1	0.86
Legend	**UC** = Use Case, **NFR** = Non-functional Requirement, **BR** = Business Rule, **DO** = Data Object, **FR** = Functional Requirement, **SF** = System Feature				

Table 5.2: Recall and Precision of the extraction

Comments about Recall

The recall for meta-model 16 (MM16) was only 97% because one instance of the logical structure, Business Rule in this case, was missed. The presentation of the business rule was completely broken due to a human error and therefore the business rule was not recognizable. In MM28, a single instance of use case was missed because one of the mandatory attributes of that instance could not be properly recognized. A single instance of use case was missed in MM30 because the method for identifying sections failed. It is important to note that this failure was caused by the failure of the SectionMapping rather than the failure of the extraction process or the meta-modeling logic.

Comments about Precision$_l$

The $precision_l$ for MM4, is less than 100% because an item was incorrectly retrieved as an instance of the functional requirement. The item was actually a part of the glossary section but had the same style and parameters as that of the functional requirement. It was not possible to make a distinction purely on the basis of structure and template.

Comments about Precision$_a$

One attribute in one instance of MM21 was not recognized properly due to the failure of SectionMapping. There were 4 instances of the meta-model 33 for which the sub-attribute (Functional Requirement) was incompletely retrieved. The authors of the document had broken the sentences and indented them to appear together. Identifying them as one chunk requires semantic analysis which is not included in the heuristics for our Mapping Queries.

The good precision and recall indicates that our framework is indeed suitable for the extraction of logical structures from software requirements documents.

5.2.4 How efficient is the extraction tool?

To measure the efficiency of the extraction tool, we recorded the amount of time it took to retrieve all the instances of a given logical structure. Table 5.3 provides extraction time in milliseconds for all meta-models. The time shown *does not* include the time taken to read the document into memory. As shown, the extraction time for most of the meta-models was less than 2 seconds which is good for practical purposes. The maximum time taken is for logical structure 14. The logical structure had 115 instances spread across 5 documents. However, the time taken (12 seconds) is still reasonable for practical purposes.

MM#	Size of Search Space	# of Instances	# of Attributes	Avg Size of Instances	Extraction Time (ms)
MM1	5239	18	11	190	1641
MM2	5239	9	6	45	453
MM3	6327	37	11	114	1141
MM4	6327	6	3	19	828
MM5	6327	21	5	76	625
MM6	3195	15	12	128	718
MM7	3195	5	3	46	156
MM8	2387	13	4	104	796
MM9	2104	12	8	79	391
MM10	2932	17	12	108	515
MM11	2932	4	3	34	218
MM12	2932	4	5	55	375
MM13	4663	29	11	121	1891
MM14	26847	115	13	120	12453
MM15	15292	5	10	3058	1375
MM16	28081	33	2	27	921
MM17	28081	296	2	23	2781
MM18	1566	8	14	182	500
MM19	5204	11	11	174	578
MM20	1905	4	6	86	359
MM21	2441	7	8	121	484
MM22	2909	6	2	87	297
MM23	8328	28	9	146	563
MM24	12715	8	10	296	1031
MM25	22667	21	11	313	797
MM26	23003	22	10	427	823
MM27	20381	30	9	156	971
MM28	14147	20	10	269	875
MM29	14154	29	10	329	953
MM30	3360	6	3	234	918
MM31	6848	14	1	76	512
MM32	2790	9	3	74	856
MM33	6757	21	8	193	1768
Total	n/a	**883**	**246**	n/a	n/a

Table 5.3: Size of Search Space, Instances of Logical Structures and Extraction Times

The time it takes to extract all instances of a logical structure depends on a lot of factors. These factors include the number of instances, the number of attributes, the size of the documents scanned (search space), the size of extracted instances and the mapping

used in the meta-model. Table 5.3 shows details about each meta-model and the instances of the meta-model. The size of search space and average size of the instances is calculated in term of number of words. The factor that most affects the extraction time is the total size of the search space and the total size of the extracted instances (extraction space). To demonstrate that, we calculated the sum of search space and extraction space for each meta-model and then sorted the extraction times according to the total size. Fig.5.2 shows the extraction times sorted by the total size(search space + extraction space). The lower x-axis denotes the meta-model number and the upper x-axis denotes the total size.

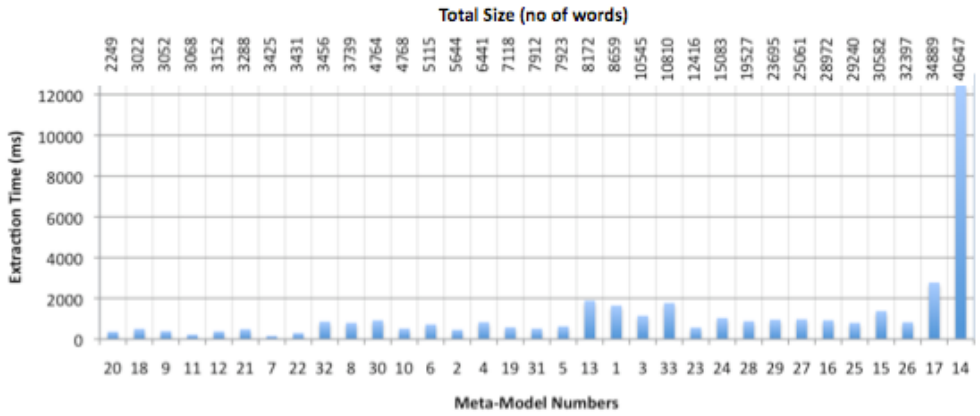


Figure 5.2: Extraction Times Sorted by Total Size (search space + extraction space)

As shown by Fig.5.2, the extraction time more or less increases as the sum of the total size of the search space and the total size of extracted instances increases. However, it is important to note that there are other factors such as the type of mappings and their associated number of parameters used in each meta-model that can affect the extraction time. The type of mapping affects the extraction time because the time taken to identify basic and complex element types is different for each element type.

5.2.5 How complex are the meta-models?

We measure the complexity of the meta-models by the number of lines used to specify them. There are many factors that determine the size of the meta-model. These factors include the number of attributes and variability in the logical structures, the extent to which details about the logical structure are modelled, the kind of Mapping and the number of Mapping parameters used. Different Mappings have different number of parameters and hence affect the size of the model differently. Each logical structure can be modelled

with a different degree of details, for example the information about the style used may or may not be modelled depending on the importance of style in uniquely identifying the logical structure. The number of false positives reduce as more details are encoded in the meta-model. Fig.5.3 shows information about the number of attributes in the meta-models and the meta-model size. The upper x-axis denotes the number of attributes. The lower x-axis denotes the meta-model numbers sorted in increasing number of attributes. The y-axis shows the size of the meta-model in terms of number of lines used to write the meta-model.

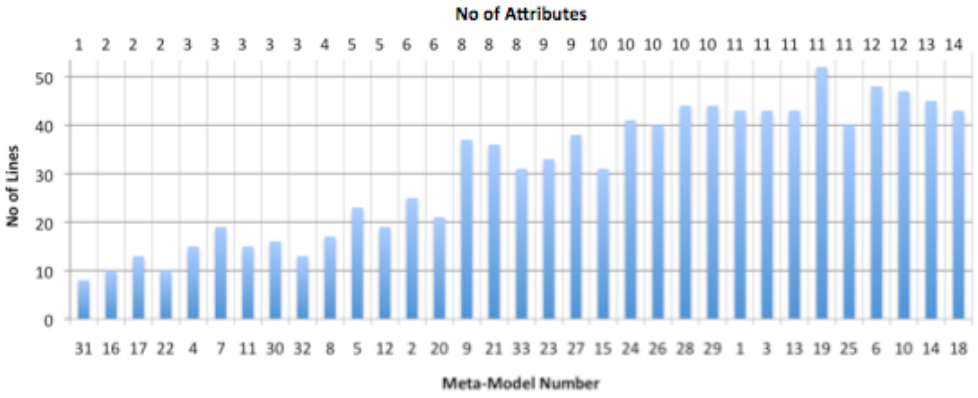


Figure 5.3: Size of the Meta-Models Sorted by No. of Attributes

Fig.5.3 shows that increasing the number of attributes does not introduce more complexity in the size of the meta-model than is introduced by the attribute itself. The maximum size of logical structures (52 Lines) is also acceptable for human viewing.

5.2.6 How do instances of logical structures vary?

We inspected each instance of each logical structure manually to observe the variability present in them. Table 5.4 shows the variability divided up into three categories – the number of attributes with variable cardinality which includes optional attributes, the number of attributes with designed physical variations which included alternate mappings and the number of accidental variations found in the *instances* of attributes. The accidental variations include the number of spelling mistakes and the number of style mistakes. There was only one meta-model, MM27, that used different attributes (flow, action-response description) to capture the same conceptual attribute.

MM#	Num of Attributes With Varaible Cardinality	Num of Attributes With Designed Variations	Accidental Variations
MM1	3	0	1
MM2	0	0	0
MM3	1	0	1
MM4	2	0	1
MM5	1	1	0
MM6	2	0	0
MM7	0	0	0
MM8	1	0	0
MM9	3	1	1
MM10	2	0	2
MM11	0	0	0
MM12	0	0	0
MM13	2	0	0
MM14	6	1	4
MM15	3	2	2
MM16	0	2	1
MM17	0	0	0
MM18	1	0	0
MM19	2	1	1
MM20	0	0	0
MM21	1	0	2
MM22	0	0	2
MM23	2	0	0
MM24	2	0	0
MM25	2	0	0
MM26	5	0	0
MM27	3	0	2
MM28	3	0	3
MM29	2	0	0
MM30	0	0	2
MM31	0	0	0
MM32	0	0	0
MM33	3	0	4
Total	**52**	**8**	**29**

Table 5.4: Variability in the Instances of Logical Structures

The table shows that there are significant number of logical variations found in the logical structures. Out of the 33 logical structures, 22 had at least one attribute that

was optional or had varying cardinality associated with it. The 33 meta-models had a total of 246 attributes, out of which 52 attributes were either optional or had a different cardinality. The variations in Mapping are not common with a total of only 8 attributes having a variation in their mapping. Out of the 8 such attributes, 3 attributes had different presentations (for example SectionMapping and TextBlockMapping), 3 had variable patterns for attribute ID (for example Numeric only and Alpha-Numeric) and 2 had variable patterns for attribute Name across instances. We also found a notable amount of accidental variations. The measurements shown in Table 5.4 are consistent with our initial observation about the subset of the data. These are significant variations that make the job of extracting logical structures non-trivial.

5.2.7 How does capturing variability affect the complexity of the meta-model?

We conducted the following experiment to investigate how capturing variability affects the complexity of the meta-model.

A single random instance of each logical structure was selected to create an initial meta-model for the logical structure. The number of lines were noted and the meta-model was used for extraction. Based on the extraction results, the meta-model was refined and the extraction was repeated until maximum recall and precision was achieved. The final meta-model was compared with the initial meta-model to note the changes in size and the refinements performed. The whole experiment was repeated twice more with one difference. For the second experiment, the best instance rather than a random instance was chosen from the instances of the logical structure. The best instance was considered to be the one that had the most number of optional attribute instances present. For the third experiment, the worst instance rather than a random instance was chosen from the instances of the logical structure. The worst instance was considered to be the one with the least number of optional attribute instances present.

Fig.5.4 shows the difference in size of the initial and final meta-model for the best, random and worst case. The difference is shown as percentage of the size of the initial meta-model. The logical structures whose instances had no variability are skipped. The lower x-axis denotes the meta-model number and the upper x-axis denotes the size of the initial meta-model. The size change was dependent on the type of changes made. The most change occurred when the initial meta-model did not contain an optional attribute and the attribute had to be added while refining the meta-model. In MM14, 3 attributes had to be made optional (an addition of "?" to 3 lines), 2 attributes had to be added (an addition of 8 lines) and one alternating mapping was added (an addition of 3 lines) in the

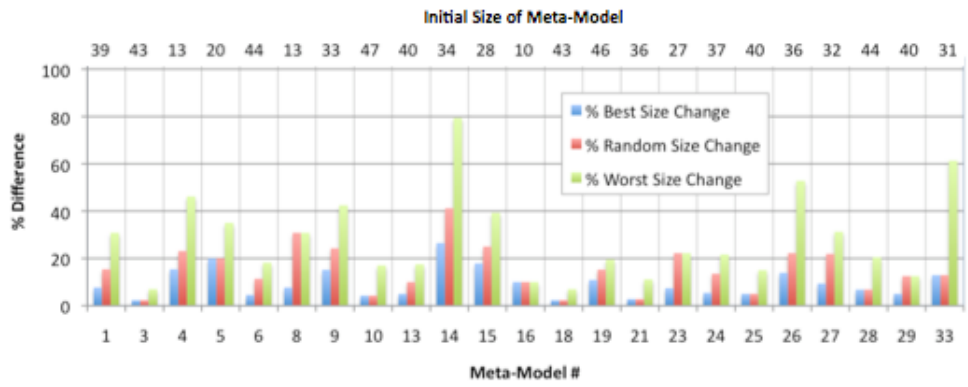


Figure 5.4: Change in Size of Meta-Models after Refinement

random case.

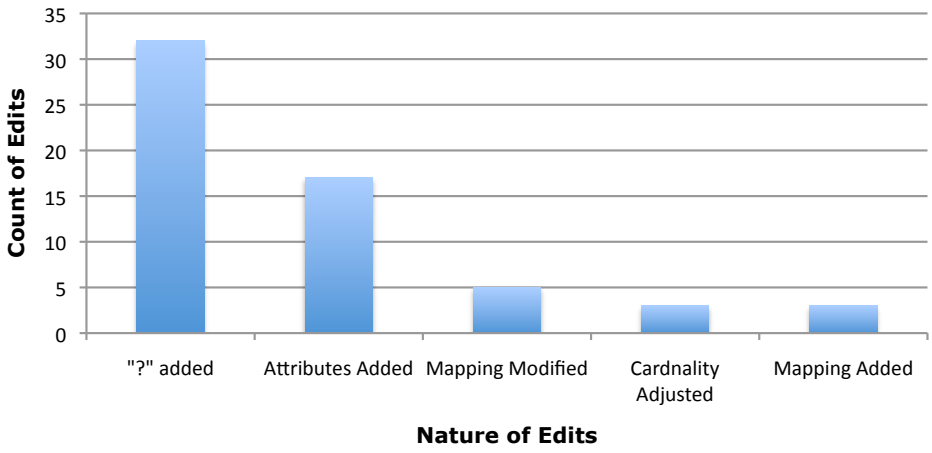


Figure 5.5: Nature of Edits during Refinement

Fig.5.5 presents the nature of edits that were made between the initial and final meta-models and their respective count. The count shown is the total count of edits across all the meta-models for the random case in the above experiment. Most of these edits were an addition of a single character "?". The character was added to make an attribute defined in the initial meta-model optional. The second most edits were made to add a missing attribute to the initial meta-model.

Fig.5.4 and Fig.5.5 show that the size of the meta-models does not change significantly

while capturing the variability and most of the edits required to capture variability are easy to make. Hence it is feasible for a user to update the meta-model in order to capture different types of variability.

5.2.8 How critical is the need for a human editable meta-model?

Building the meta-model from scratch requires a significant understanding of the framework. The future work will focus on the induction of these meta-models from some examples. As explained in Sec. 1.2.1, it would not be possible to capture all of the variability by using only a few example instances. It would be a lot easier to manually modify the induced template to specify which attributes are optional (by just adding "?") and variability in the presentations (by specifying another mapping type). To motivate the case for a human readable and editable meta-model, we investigated that if there were an induction system for meta-models which did not allow editing, how many instances would be required to completely capture the variability. The following experiment was conducted for the investigation.

A random instance was picked from the set of all instances of a given logical structure. The instance was used to create the meta-model of the logical structure. The meta-model was used to extract other instances of the logical structure and the precision and recall was noted. The single random instance chosen did not capture any variability present in the instances and therefore the recall and precision were less than the maximum possible. Next, two random instances were selected from the set of all instances of that logical structure to create the meta-model and the extraction was repeated. If precision and recall were not equal to the maximum possible, the meta-model was further refined. The process was repeated picking one more instance than on the previous occasion until the meta-model was good enough to extract instances with maximum precision and recall. The number of instances it took to get to the final meta-model was noted. The whole process was repeated five times for each logical structure in our set of 33.

Fig.5.6 shows the median number of instances taken to get to the final meta-model over 5 iterations. The lower x-axis shows the meta-model number and the upper x-axis shows the total number of instances for each logical structure.

As shown by Fig.5.6, some meta-models, such as MM14, MM26, MM28, require a significant number of example instances to fully capture the variability present in their instances. In the case of MM26 and MM28, the number of example instances constitute almost 50% of the total number of instances. It may not be feasible to provide half the number of desired instances as examples in order to extract the rest of them. It should be

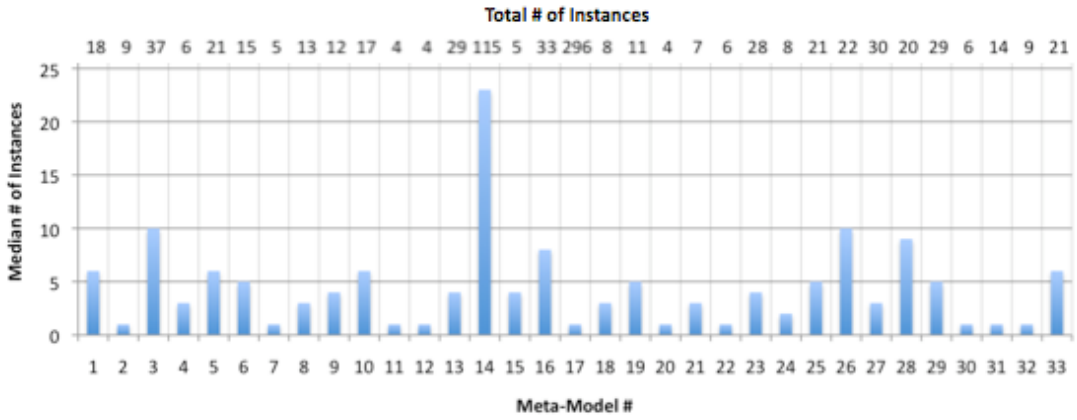


Figure 5.6: Median % of Instances

noted that the accidental physical variations were ignored in the above experiment. If we include the accidental variations, the median number of required instances would likely go up. Given that most of the variability is due to optional attributes which can be considered expert knowledge, it would be more feasible to induce the meta-model from a couple of examples and then edit it to incorporate the variability.

5.3 Threats to Validity

We now discuss the threats to validity of the experiments conducted, the results obtained and the limit to generalization. In particular, we discuss the threats to *internal* and *external* validity and the measures taken to minimize such threats.

5.3.1 Threats to Internal Validity

Threats to internal validity relate to unexpected sources of bias that may compromise the design and analysis of the study [40]. The main threats to internal validity of our study are the methods used to record measurements. The recall and precision may be influenced by the inclusion of a false negative, which makes the recall appear higher and the exclusion of a false positive which makes the precision appear higher. To minimize these threats, the results of the experiments were manually verified by looking at highlighted and missed instances in the documents and manually scanning the output XML to verify extraction

of attributes.

5.3.2 Threats to External Validity

Threats to external validity relate to the extent to which the conclusions can be generalized [40]. We identified a set of 15 mappings that were enough to model logical structures in the documents we evaluated. However, there is no claim that the same set of mappings will be enough to model all logical structures belonging to all sorts of documents. The precision and recall of our results was based on these mappings. Therefore, we claim that the same level of precision and recall can only be expected if logical structures are modelled using the mappings we identified.

Another threat to external validity relates to the design of the study. Ideally, the data set for the study should be divided into two separate sets – the training set and the evaluation set. In the study, we used a randomly chosen subset of the data set to develop our technique but presented results over the entire data set. However, results for each individual item in the data set were presented including those that were used for the initial development and those that were used later. The results show an equal amount of precision and recall between the two, indicating that the identified mappings and the extraction process were applicable to the new encountered documents.

Chapter 6

Conclusion

The ability to recognize and extract logical structures such as use cases, functional and non requirements, system features, etc. from software requirements documents is critical for the coherence of sophisticated requirements management and traceability tools and general purpose rich-text editors. The thesis has identified a set of features necessary for any practical logical extraction system. The proposed framework satisfies the requirements identified in Sec.1.2.1. The thesis has provided evidence that logical structures and variability in their templates can be naturally and concisely modelled using the mappings that were identified. It has also shown that, using the extraction tool included in the framework, different logical structures can be extracted with high precision and recall, each close to 100%. The performance of the extraction tool is acceptable for fast extraction of logical structures from documents with extraction times ranging from a few milliseconds to a few seconds.

The performance of the framework is based on the performance of several key components. We now discuss their limitations and future work that is needed to address these challenges.

6.1 Limitations and Future Work

One of the basic assumptions is that a consistent template with slight variations is usually followed across a single project allowing us to build a meta-model from very few examples and extract the rest of the instances. The framework would be unable to handle situations where no template is followed across the project or when documents are extremely incon-

sistent in presentation. However, we believe that such a case is unlikely in the professional environment of industry.

6.1.1 Document Queries

We assume that document queries can be written for all presentations. If a document query cannot extract elements from the document with significant accuracy, our system may not be as applicable. The performance of document queries is based heavily on the correctness of the identification of complex document elements. As discussed in the evaluation, there are cases when the heuristics for identification of complex document elements fail. There are two broad reasons for the failure of complex element identification. First, the document may not have enough style information to distinguish between different elements. Second, the style information presented did not fit the rules defined by the heuristics.

In the first case, the document can be considered free running text and information extraction from unstructured text becomes a separate problem altogether. The problem can be solved by using a text segmentation approach that analyzes the natural language and determines the boundaries of semantically related information [41]. A probabilistic model is usually built using domain specific examples to classify passages in the text. However, such a solution will not be as robust and would require extensive training on different examples for different logical structures.

In the second case, the identification rules that use the style and structure information can be learnt from a large set of example elements rather than being hand coded. Some work has already been done to determine elements in HTML documents that are not defined using proper tags [42, 43]. A probabilistic model is built from examples and is used to extract structures from the text. The technique in [44] incorporates domain specific vocabulary along with structural cues to improve section recognition in medical reports.

In future, these techniques can replace the heuristics for the identification of complex document elements. However, because of the separation of queries from the identification of document elements and meta-models from queries in our system, the rest of the framework will remain the same.

6.1.2 Learning Meta-Models from Examples

Although Clafer provides an easy and powerful notation to write meta-models for extraction, it may not be feasible for non-technical users to write models directly. Therefore, it

is important to have a mechanism where a user tags examples and the meta-model is built from those examples. Ideally, the tool should learn everything from tagged examples but this may not be practical because of two reasons.

1. Such a tool would require a lot of examples to create the model, which is not practical for real situations. The users cannot be required to provide many examples, as it would take a lot of time and defeat the whole idea of automation.

2. To correctly model certain aspects, like an attribute being mandatory or optional, varying examples covering all possible situations must be tagged and used. Choosing a representative sample is a difficult task that requires deep knowledge about the tool and more importantly extra time.

Therefore we envision a tool which can do three things: learn a meta-model from a few examples, improve the meta-model if more examples are input, and provide a natural interface for users to edit the meta-model. Using Clafer for modelling has the advantage of providing a concise and simple notation to allow edits to the inferred model.

In our envisioned tool, the user would begin by specifying the logical structure for which they are training the system and highlight attributes in the documents. The tool would automatically build a meta-model with appropriate attribute identifiers and mappings. An inspiration for such a tool is Thresher [45] which is a browser plug-in used to learn and extract required information from arbitrary Websites. The induction of the meta-model from examples involves determining the attributes, attribute cardinality, alternate mapping groups, and inducing pattern parameters, such as sectionTitlePattern, involved in the mappings. Techniques similar to [46] can be used to determine the attributes and mapping groups. A wrapper induction system like WHISK can be used to identify the *pattern* parameters. While wrapper induction is not suitable for capturing complete logical structures, WHISK will be useful in determining single line patterns.

APPENDICES

Appendix A

The following sub-appendices provide details about the Mappings we identified and used in the experiments. Each sub-appendix provides the description of the Mapping, its Clafer definition and the functionality of the associated query. The parameters of the queries are shown with the query. The underlined parameters are mandatory parameters for the queries and must always be given. Other parameters are optional and may or may not be given.

A.1 ParagraphMapping

The ParagraphMapping is used to map to paragraphs in the documents. A paragraph may contain multiple lines of text.

Clafer Definition:

```
ParagraphMapping : Mapping
    style -> LSStyle ?
    pattern : String ?
```

ParagraphQuery *(scope, style, pattern)*

If no parameter is provided, the query extracts all the sections in the given *scope*. If *pattern* is specified, only the paragraphs that match the given pattern will be matched. If *style* is specified, only the paragraphs matching the given style will be matched.

A.2 ParagraphPartMapping

ParagraphPartMapping is used when some attribute maps to a part of the paragraph. The pattern must be of the form expression regex1 {regex2} regex3 where regex1-3 are optional.

Clafer Definition:

```
ParagraphPartMapping : Mapping
    pattern : String
```

ParagraphPartQuery *(scope, pattern)*

The paragraphpart query applies the given pattern to all the paragraphs in the scope. The fragment of each matching paragraph that matches the given regex2 is extracted.

A.3 TextBlockMapping

It is used to map to text-block structures in the documents. The textblock structure looks like the following:

Identifier [delimiter] Value

The following is an example of a text-block where 'name' is the identifier and 'Rehan' is the value:

Name : Rehan

Clafer Definition:

```
TextBlockMapping : Mapping
    identPattern : String?
    identText : String?
    delimiter : String
```

TextBlockQuery *(scope, identPattern, identText, delimiter)*

The query matches all the text-blocks of the above mentioned form. One of identPattern or identText must be provided. The delimiter cannot be part of the identPattern or the identText. The query matches all text-block whose identifier matches the given identPattern or identText.

A.4 SectionMapping

It is used to map to sections or subsections. For our purposes, we dont make a distinction between sections and sub-sections. We consider sub-sections to be sections within another section.

Clafer Definition:

```
SectionMapping : Mapping
    sectionTitlePattern : String  ?
    sectionTitleText : String  ?
    sectionTitleStyle ->LSStyle  ?
```

SectionQuery *(scope, sectionTitlePattern, sectionTitleText, sectionTitleStyle)*

If no parameters are given, the section query extracts all the sections in the given scope. If sectionTitlePattern is provided, only the sections whose title matches the given pattern are extracted. If sectionTitleText is provided, only the sections whose title matches the given text are extracted. Only one of these two parameters can be given at one time. If sectionTitleStyle is provided, only the sections whose title matches the given style are extracted.

A.5 SectionTitleMapping

It is used to match either the whole section title or part of the section title. The pattern must be of the form expression regex1 {regex2} regex3 where regex1-3 are optional.

Clafer Definition:

```
SectionTitleMapping : Mapping
    identPattern : String?
```

SectionTitleQuery *(scope, identPattern)*

If the identPattern is provided, the section-title query applies the given identPattern to the title of each section in the scope. The fragment of each matching title that matches the given regex2 is extracted. If no identPattern is provided, the query extracts the title of each end every section in the given scope.

A.6 ListMapping

It is used to match enumerated or bulleted lists in the documents. A list with a title is quite like a section. The difference is that each element in the list must be either an enumerated or bulleted paragraph. In the section, there is no such restriction.

Clafer Definition:

```
ListMapping : Mapping
    listTitlePattern : String ?
    listTitleText : String ?
    listIdentLevel : Integer ?
    listTitleStyle ->LSStyle ?
```

ListQuery *(scope, listTitlePattern, listTitleText, listTitleStyle)*

If no parameters are given, the list query extracts all the lists in the given scope. If listTitlePattern is provided, only the lists whose title matches the given pattern are extracted. If listTitleText is provided, only the lists whose title matches the given text are extracted. Only one of these two parameters can be given at one time. If listTitleStyle is provided, only the lists whose title matches the given style are extracted. The parameter listIndentLevel can be used to extract sublists within the lists.

A.7 DocumentMapping

The DocumentMapping is used to map to entire documents.

Clafer Definition:

```
DocumentMapping : Mapping
    namePattern : String ?
```

DocumentQuery *(scope, namePattern)*

If namePattern is specified, the document-query matches only those documents whose name matches that pattern. This may be used to restrict the search to a certain set of documents in the repository

A.8 DocumentTitleMapping

It is used to match either the whole document title or part of the document title. The pattern must be of the form expression regex1 {regex2} regex3 where regex1-3 are optional.

Clafer Definition:

```
DocumentTitleMapping : Mapping
    style->LSStyle?
    identPattern : String?
```

DocumentTitleQuery *(scope, style, identPattern)*

The search for document titles is limited to the first page of the document. Each paragraph in the first page is scanned and the ones with heavier styles are selected as document title. If the identPattern is provided, the document-title query applies the given identPattern to each paragraph in the title of each document found in the provided scope (which may be repository or a document) and extracts the fragment of each matching paragraph that matches the given regex2. If style is provided, only the paragraphs in the title matching the given style are considered for extraction. If no pattern is provided, all the filtered paragraphs in the title page are extracted.

A.9 CellMapping

The CellMapping is used to map to table-cells in the documents. A cell may contain multiple paragraphs, graphics, lists, sections etc.

Clafer Definition:

```
CellMapping : Mapping
    colIndex : Integer ?
    rowIndex : Integer ?
    contentStyle->LSStyle ?
```

CellQuery *(scope, style, pattern)*

If no parameter is provided, the query extracts all the cells of all the tables in the given scope. If colIndex or rowIndex is provided, only the cells that have the specified column index or row index are extracted. If contentStyle is specified, only cells whos content has the specified style are matched.

A.10 TableMapping

The TableMapping is used to map to complete tables in the documents. The table heading can be provided. The following two forms of table heading are recognized:

Table Heading

X	X
X	X
X	X

(a) Outside the Table

Table Heading	
X	X
X	X
X	X

(b) In the first Row

Clafer Definition:

```
TableMapping : Mapping
    headingText : String ?
    headingPattern : String ?
    headingStyle -> LSStyle ?
```

TableQuery *(scope, headingText, headingPattern, headingStyle)*

If no parameters are given, the table-query extracts all the tables in the given scope. If headingPattern is provided, only the tables whose heading matches the given pattern are extracted. If headingText is provided, only the tables whose heading matches the given text are extracted. Only one of these two parameters can be given at one time. If headingStyle is provided, only the tables whose heading matches the given style are extracted.

A.11 HCellBlockMapping

The HCellMapping is used to map to horizontal blocks of two cells in the tables. A horizontal cell block structure is of the following form:

Identifier	Value

```
HCellBlockMapping : Mapping
    identText : String ?
    identPattern : String ?
    identStyle->LSStyle ?
```

Clafer Definition:

HCellBlockQuery *(scope, identText, identPattern, identStyle)*

If identPattern is provided, only the cell-blocks whose identifier matches the given pattern are extracted. If identText is provided, only the cell-blocks whose identifer matches the given text are extracted. Only one of these two parameters can be given at one time and one of these must be given. If identStyle is provided, only the cell-blocks whose identifier matches the given style are extracted.The scope for cell-block query can contain multiple tables. In that case, matching cell-blocks from all tables will be extracted.

A.12 VCellBlockMapping

The VCellMapping is used to map to vertical blocks of two cells in the tables. A vertical cell block structure is of the following form:

Identifier
Value

Clafer Definition:

```
VCellBlockMapping : Mapping
    identText : String ?
    identPattern : String ?
    identStyle->LSStyle ?
```

VCellBlockQuery *(scope, identText, identPattern, identStyle)*

If identPattern is provided, only the cell-blocks whose identifier matches the given pattern are extracted. If identText is provided, only the cell-blocks whose identifer matches the given text are extracted. Only one of these two parameters can be given at one time and one of these must be given. If identStyle is provided, only the cell-blocks whose identifier matches the given style are extracted. The scope for cell-block query can contain multiple tables. In that case, matching cell-blocks from all tables will be extracted.

A.13 ColumnMapping

The ColumnMapping is used to map to columns tables in the documents. The following two forms of columns are recognized:

X	X	X	X	X
X	**Identifier**			X
X	val	val	...	X
X	val	val	...	X

(a)

X	X	X	X
X	**Identifier**		X
X	val		X
X	val		X

(b)

Clafer Definition:

```
ColumnMapping : Mapping
    identText : String ?
    identPattern : String ?
    identStyle ->LSStyle ?
```

ColumnQuery *(scope, identText, identPattern, identStyle)*

If identPattern is provided, only the columns whose identifier matches the given pattern are extracted. If identText is provided, only the columns whose identifer matches the given text are extracted. Only one of these two parameters can be given at one time and one of these must be given. If identStyle is provided, only the columns whose identifier matches the given style are extracted. The scope for column-query can contain multiple tables. In that case, matching columns from all tables will be extracted.

A.14 RowMapping

The RowMapping is used to map to rows tables in the documents. The following two forms of rows are recognized:

Clafer Definition:

RowQuery *(scope, identText, identPattern, identStyle)*

If identPattern is provided, only the rows whose identifier matches the given pattern are extracted. If identText is provided, only the rows whose identifer matches the given text

X	X	X	X	X
Identifier	val	val	val	val
X	X	X	X	X
X	X	X	X	X

(a)

X	X	X	X	X
Identifier	val	val	val	val
	val	val	val	val
X	X	X	X	X

(b)

```
RowMapping : Mapping
    identText : String  ?
    identPattern : String  ?
    identStyle->LSStyle  ?
```

are extracted. Only one of these two parameters can be given at one time and one of these must be given. If identStyle is provided, only the rows whose identifier matches the given style are extracted. The scope for row-query can contain multiple tables. In that case, matching rows from all tables will be extracted.

A.15 GraphicMapping

It is used to match graphic objects such as clip-art and picture in the documents.

Clafer Definition:

```
GraphicMapping : Mapping
    caption : String?
```

GraphicQuery *(scope, caption)*

If caption is not provided, the query extracts all the graphic objects in the given scope. If caption is provided, only the graphics whose caption matches the given caption are extracted.

Bibliography

[1] *INCOSE Requirements Management Tools Survey*, 2010 (accessed Nov 1, 2010). `http://www.incose.org/ProductsPubs/products/rmsurvey.aspx`. 1

[2] R.R. Sud and J.D. Arthur. Requirements management tools: A quantitative assessment. Technical Report TR-03-10, Virginia Tech, 2003. 1

[3] P Zielczynski. *Requirements management using ibm®rational®requisitepro®*. IBM Press, 2007. 1, 12

[4] L. Goldin and D.M. Berry. Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engineering*, 4:375–412, 1997. 10.1023/A:1008617922496. 1

[5] F. Fabbrini, M. Fusani, S. Gnesi, and G. Lami. An Automatic Quality Evaluation for Natural Language Requirements. In *in Proceedings of the Seventh International Workshop on RE: Foundation for Software Quality (REFSQ'2001*, volume 1, 2001. 1, 12

[6] M. Osborne and C.K. MacNish. Processing natural language software requirement specifications. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pages 229 –236, April 1996. 1

[7] Daniel M. Berry, Krzysztof Czarnecki, Michał Antkiewicz, and Mohamed AbdelRazik. Requirements determination is unstoppable: An experience report. In *Requirements Engineering*, 09/2010 2010. 2

[8] P. Kruchten. *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley Professional, December 2003. 3

[9] E. Hull, K. Jackson, and J. Dick. Doors: A tool to manage requirements. *Requirements Engineering*, pages 173–189, 2005. 3

[10] A. Cockburn. *Basic use case template*, 2010 (accessed October 17, 2010). http://alistair.cockburn.us/Basic+use+case+template. 5

[11] *Use-Case Template*, 2010 (accessed October 17, 2010). http://publib.boulder.ibm.com/infocenter/pim/v6r0m0/index.jsp?topic=/com.ibm.wpc.sar.doc/wpc_ref_usecasetemp.html. 5

[12] *Introducing the Office (2007) Open XML File Formats*, 2010 (accessed October 17, 2010). http://msdn.microsoft.com/en-us/library/aa338205(office.12).aspx. 6, 17

[13] *Convert HTML to DOC*, 2010 (accessed October 17, 2010). http://www.eprintdriver.com/to_word/HTML_to_Word_Doc.html. 6

[14] *Nitro PDF to Word*, 2010 (accessed October 17, 2010). http://www.pdftoword.com/. 6

[15] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in Clafer: Mixed, specialized, and coupled. In *3rd International Conference on Software Language Engineering*, Eindhoven, The Netherlands, 10/2010 2010. 9, 11

[16] W.M. Wilson, L.H. Rosenberg, and L.E. Hyatt. Automated analysis of requirement specifications. *Software Engineering, International Conference on*, 0:161, 1997. 12

[17] L. Mich. NL-oops: from natural language to object oriented requirements using the natural language processing system lolita. *Nat. Lang. Eng.*, 2(2):161–187, 1996. 12

[18] S. Nanduri and S. Rugaber. Requirements validation via automated natural language parsing. *Hawaii International Conference on System Sciences*, 0:362, 1995. 12

[19] V. Gervasi and B. Nuseibeh. Lightweight validation of natural language requirements: a case study. In *Requirements Engineering, 2000. Proceedings. 4th International Conference on*, pages 140 –148, 2000. 12

[20] A. Sinha, S.M. Sutton Jr., and A. Paradkar. Text2test: Automated inspection of natural language use cases. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:155–164, 2010. 12

[21] *Reqtify*, 2010 (accessed October 17, 2010). http://www.geensoft.com/en/article/reqtify. 12

[22] N. Kushmerick. *Wrapper induction for information extraction*. PhD thesis, University of Washington, 1997. Chairperson-Weld, Daniel S. 14

[23] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118:2000, 2000. 14

[24] S. Soderland. Learning information extraction rules for semi-structured and free text. *Mach. Learn.*, 34(1-3):233–272, 1999. 14, 23

[25] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 190–197, New York, NY, USA, 1999. ACM. 14, 16

[26] S. Zheng, R. Song, J. Wen, and C. L. Giles. Efficient record-level wrapper induction. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 47–56, New York, NY, USA, 2009. ACM. 14, 15

[27] H. Garcia-Molina Cho, J. Hammer, H. Garcia-molina, Cho J., R. Aranha, and A. Crespo. Extracting semistructured information from the web. In *In Proceedings of the Workshop on Management of Semistructured Data*, pages 18–25, 1997. 14

[28] S. Kuhlins and R. Tredwell. Toolkits for generating wrappers. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 184–198, London, UK, 2003. Springer-Verlag. 14

[29] H. Chun-Nam. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Papers from the 1998 Workshop on AI and Information Integration*. AAAI Pres, 1998. 15

[30] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 175–186, New York, NY, USA, 2001. ACM. 15

[31] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of framework-specific modeling languages. *IEEE Transactions on Software Engineering*, 35:795 – 824, 11/2009 2009. 17, 18

[32] *Eclipse Language IDE*, 2010 (accessed November 30, 2010). `http://www.eclipse.org/home/categories/index.php?category=ide`. 17

[33] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. 18

[34] *Code Samples and Apps: Applets*, 2010 (accessed November 30, 2010). `http://java.sun.com/applets/`. 18

[35] M. Antkiewicz, T.T. Bartolomei, and K. Czarnecki. Fast extraction of high-quality framework-specific models from application code. *Automated Software Engineering*, 16:101 – 144, 03/2009 2009. 19

[36] R.C. Angell, G.E. Freund, and P. Willett. Automatic spelling correction using a trigram similarity measure. *Information Processing & Management*, 19(4):255 – 261, 1983. 35

[37] R. Ferber. Research by convenience. *Journal of Consumer Research: An Interdisciplinary Quarterly*, 4(1):57–58, June 1977. 40

[38] *Use Cases Database (UCDB)*, 2010 (accessed October 17, 2010). `http://www.se.cs.put.poznan.pl/knowledge-base/software-projects-database/use-cases-database-ucdb`. 40

[39] J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel. Performance measures for information extraction. 1999. 43

[40] B.A. Kitchenham, S.L. Pfleeger, L.M. Pickard, P.W. Jones, D.C. Hoaglin, K. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28:721–734, 2002. 53, 54

[41] J.P. Yamron, I. Carp, L. Gillick, S. Lowe, and P. van Mulbregt. A hidden markov model approach to text segmentation and event tracking. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 1, pages 333 –336 vol.1, May 1998. 56

[42] K. Lerman, L. Getoor, S. Minton, and C. Knoblock. Using the structure of web sites for automatic segmentation of tables. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 119–130, New York, NY, USA, 2004. ACM. 56

[43] M. Yoshida and K. Torisawa. A method to integrate tables of the world wide web. 2001. 56

[44] P.S. Cho, R.K. Taira, and H. Kangarloo. Automatic section segmentation of medical reports. In *Proceedings of the 2003 AMIA Annual Symposium*, pages 155–159. American Medical Informatics Association, 2003. 56

[45] A. Hogue and D. Karger. Thresher: Automating the unwrapping of semantic content from the World Wide Web. In *in Proceedings of the Fourteenth International World Wide Web Conference*, pages 86–95, 2005. 57

[46] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE 2011*, To Appear. 57